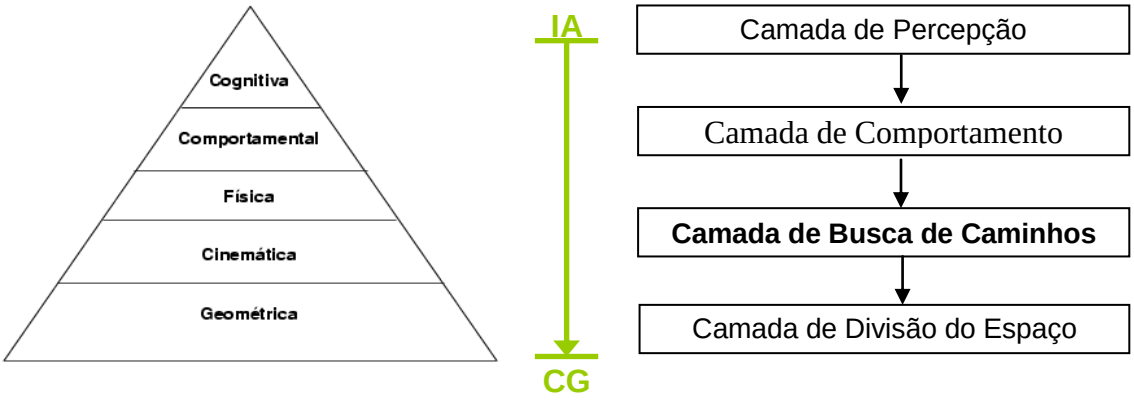
Universidade Federal de Santa Maria
Departamento de Eletrônica e Computação
Prof. Cesar Tadeu Pozzer
Disciplina: Computação Gráfica Avançada
pozzer@inf.ufsm.br
20/10/2007

Planejamento de Caminho



Na camada de **Busca de caminhos**, o módulo de IA utiliza as informações computadas pela Camada de Divisão do Espaço para buscar caminhos entre pontos de origem e destino. Locomover-se no espaço do jogo é uma ação fundamental aos personagens não-jogadores em qualquer gênero de jogos. Assim, a busca de caminhos deve ser implementada de maneira muito eficiente, pois será executada muitas vezes por vários personagens durante o jogo [Gilliard Loppes].

1. Teoria dos Grafos

Ver material de grafos em Estruturas de Dados.

2. Path-Planning

Como apresentado na seção anterior, grafos de navegação podem ser utilizados pelos agentes para planejar caminhos entre duas posições no ambiente. Nesta seção serão abordados aspectos práticos que visam solucionar problemas que geralmente são encontrados quando se aplica *path-finding* em ambientes não hipotéticos, como ocorre em qualquer jogo.

2.1. Construção de um grafo de navegação

O grafo de navegação é uma estrutura de repartição do espaço do jogo utilizada pelos algoritmos de busca para determinar o caminho, possivelmente o melhor, que une dois pontos. Uma vez que existem várias formas de representar a geometria do ambiente do jogo, como blocos (*tiles* ou células), vetor ou poligonal, também existem várias estratégias para converter a informação espacial pertinente em uma estrutura em formato de grafo. A abordagem poligonal geralmente é adotada em jogos do tipo FPS, onde o cenário é construído inteiramente com polígonos.

2.1.1. Tiles

Este tipo de abordagem é muito comum em jogos do tipo RTS e jogos de guerra. Geralmente são grafos grandes e complexos, organizados por meio de quadrados ou hexágonos. Cada nó do grafo representa o centro de cada célula (Figura 1). As arestas representam a vizinhança de cada célula. Geralmente cada célula está associada a um tipo de terreno, que pode ser plano, macio, movediço, montanhoso, ou até mesmo água. Essa informação da composição do terreno é usada como peso de cada aresta do grafo de navegação. Assim, o algoritmo de busca de caminho leva em consideração, além da distância, esses fatores, de modo a evitar que as unidades do jogo trafeguem por regiões que apresentem maiores empecilhos.

O grande problema esta abordagem é que o número de vértices e arestas podem se tornar rapidamente muito elevados. Para um mapa com 100 x 100 células, tem-se 10.000 nós e 78.000 arestas. Como os jogos do tipo RTS geralmente possuem muitas unidades (acima de centenas), além de permitirem vários times jogando simultaneamente, o custo de processamento com *path-planning* pode se tornar proibitivo.

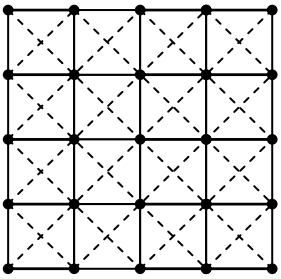


Figura 1: Exemplo de um grafo de navegação em *tiles*

2.1.2. Pontos de Visibilidade

A criação de um grafo de navegação baseado em Pontos de Visibilidade (POV – *Points of Visibility*) consiste na adição de nós, em um processo geralmente feito a mão pelo game designer, em pontos importantes do ambiente de modo que cada nó tenha pelo menos uma linha de visada para outro nó. Posicionados cuidadosamente, os nós do grafo produzirão um grafo que conecta todas as áreas importantes da geometria do cenário (Ver Figura 2).

A concepção deste grafo permite que sejam facilmente adicionados novos nós que possam representar pontos de interesse para os agentes, como por exemplo posições que são boas para se ficar de tocaia, para se esconder, dentre outras. O problema é que para um cenário grande, pode ser muito trabalhoso a definição dos pontos do grafo. Além disso, por ser um trabalho manual, impede que cenários sejam gerados de forma randômica durante a execução do jogo. Esse é um motivo que faz com que muitos jogos não possuam a opção de geração de mapas aleatórios.

Como o grafo é definido por poucos pontos, ou seja, cada nó representa uma grande região, este tipo de grafo é conhecido como "grosseiramente granulado" – *coarsely granulated*. São grafos compactos, que gastam pouca memória e permitem buscas rápidas. Porém, possuem várias limitações.

Se o jogo restringe o movimento dos agentes somente sobre as arestas do grafo, como ocorre no jogo Pac-man, esta solução é a escolha perfeita. Entretanto, se o grafo é projetado para um jogo onde os agentes têm maior liberdade de movimentos, diversos problemas podem ser percebidos. Por exemplo, em jogos do tipo RTS o

jogador pode selecionar unidades e movê-las para qualquer local do cenário. Para realizar esta tarefa, baseado em um grafo de visibilidade, a IA realiza uma série de tarefas:

1. Procura o nó visível A mais próximo da posição do NPC;
2. Procura o nó visível B mais próximo da posição destino
3. Usa algum algoritmo **(A*)** para encontrar o caminho (conjunto ordenado de nós) de menor custo entre A e B;
4. Move o NPC para o nó A;
5. Move o NPC sob o caminho calculado no passo 3;
6. Move o NPC do nó B até a posição destino.

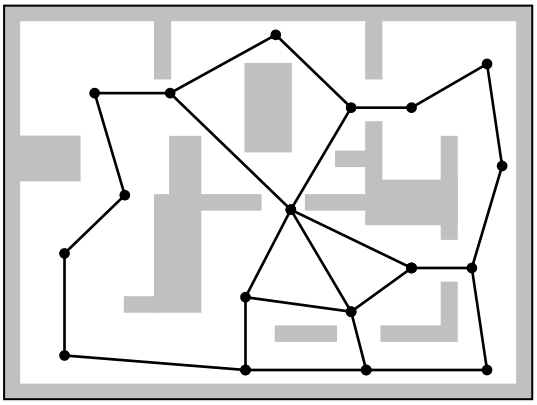


Figura 2: Pontos de Visibilidade de um Grafo de navegação

Se esses passos forem executados em um grafo com poucos nós, como o mostrado na Figura 2, situações muito estranhas podem ocorrer, como mostrado na Figura 3.

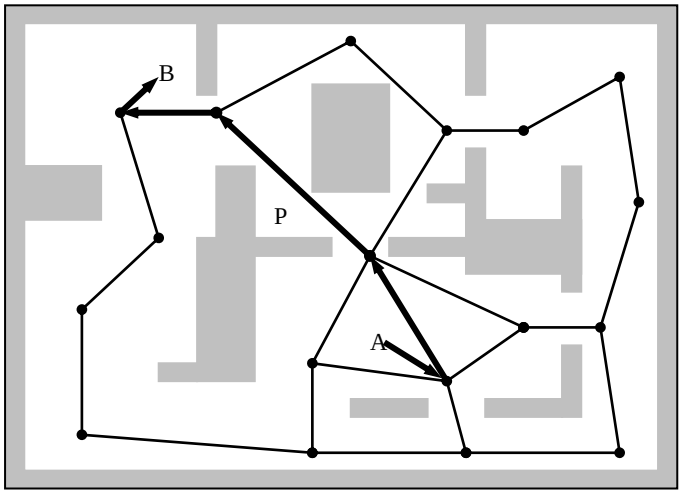


Figura 3: Processo de seleção de caminho entre A e B; exemplo de nó P invisível ao grafo de navegação.

Devido a uma malha de nós muito esparsa, o agente pode realizar movimentos em zig-zags para chegar ao destino. Esses movimentos não são naturais e degradam o aspecto de IA do jogo. Além disso, uma vez que a definição dos nós é feita manualmente, podem existir locais do cenário que não tem nenhum nó visível, como representado pela posição P do grafo da Figura 3 especialmente quando o cenário for grande e complexo.

2.1.3. Geometria Expandida

Esta técnica visa suprir os problemas da técnica de POV. Se o ambiente do jogo é construído a partir de polígonos, é possível usar a informação presente em cada forma para gerar automaticamente o grafo de visibilidade, independente do tamanho do cenário. Para isso, inicialmente expandem-se os polígonos por um valor proporcional ao raio do círculo envolvente dos agentes do jogo (Figura 4). Os vértices que definem a

geometria expandida são então usados como nós do grafo de navegação. Após isso, faz-se uso de um algoritmo para determinar quais pontos são visíveis a partir dos outros, para então definir as arestas.

Uma vez que o polígono expandido é proporcional ao raio do círculo envolvente dos agentes, estes podem planejar suas trajetórias sem nuca tocar nos obstáculos.

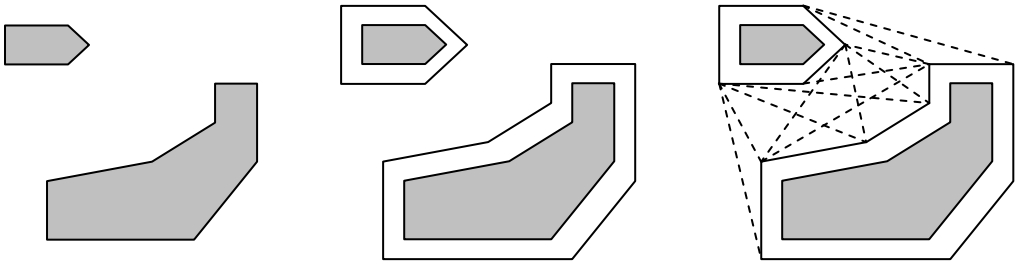


Figura 4: Criação dos Pontos de visibilidade usando Geometria Extendida

2.1.4. NavMesh

A estratégia chamada *Navmesh* está crescendo em popularidade entre os desenvolvedores de jogos. Consiste na geração de uma rede de polígonos convexos para descrever as áreas que podem ser percorridas pelos personagens. Um polígono convexo tem como característica de se traçar qualquer aresta dentro de seu espaço sem nenhuma interseção. Assim, cada nó do grafo representa um espaço convexo, ao invés de um único ponto. Isso traz enorme eficiência aos algoritmos de *path-finding*, visto que operam sobre um número reduzido de nós, especialmente quanto comparados com uma abordagem de *tiles*. Para jogos baseados em polígonos, como no caso dos jogos FPS, pode-se usar esta técnica para particionar as áreas acessíveis do mapa de forma automática.

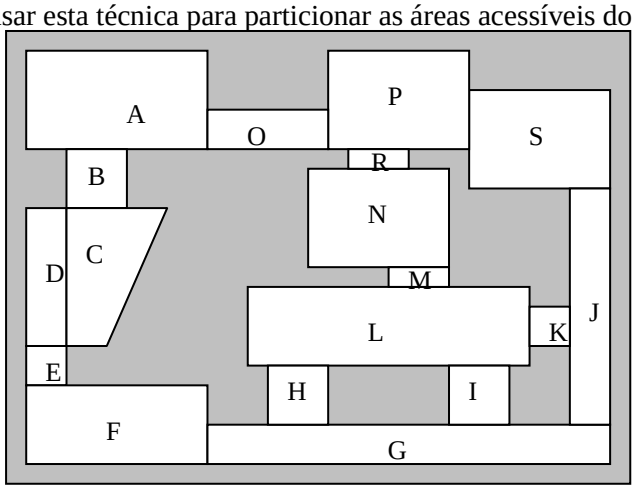


Figura 5: *Navmesh* para um cenário de jogo FPS

2.2. Refinamento de Grafos

Problemas como caminhos com baixa qualidade (granularidade) e posições inacessíveis, como apresentado na Seção 2.1.2, podem ser resolvidos aumentando-se a granularidade do grafo de navegação. A Figura 6 apresenta o grafo da Figura 2 com alta granularidade. Este exemplo de grafo é muito similar ao grafo de grade (*tiles*), apresentado na Seção 2.1.1, e conseqüentemente apresenta os mesmos desafios de implementação.

Uma estratégia para a geração desta malha refinada de nós é o uso de um algoritmo de preenchimento (*Flood fill*), semelhante aos algoritmos para pintura de polígonos. O algoritmo recebe como dado de entrada uma semente, que é um nó inserido em qualquer local válido do cenário. A partir deste ponto, o algoritmo expande um grafo (nós e arestas) em todas as direções disponíveis, até que toda área esteja coberta (Figura 7). Após o preenchimento, o designer pode adicionar ou remover nós de interesse para obter o resultado desejado. Para garantir que o agente não irá tocar os obstáculos, considera-se o raio do círculo envolvente dos agentes na determinação se um nó deve ser inserido ou não no grafo.

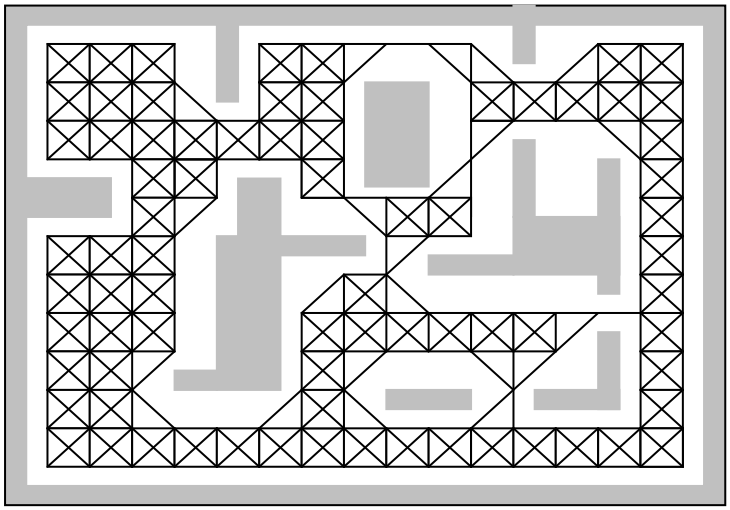


Figura 6: Grafo de navegação com alta granularidade

Outra tarefa que o designer pode realizar é a inserção de itens ao grafo de navegação. Os itens podem ser qualquer coisa que os agentes podem capturar durante o jogo, como munição, kit médico, ouro, madeira, minerais, dentre outros. Recomenda-se que estes itens estejam próximos as principais rotas que o agente realiza. Isso possibilita que o agente passe mais tempo focado no propósito do que ficar procurando pelos itens no cenário.

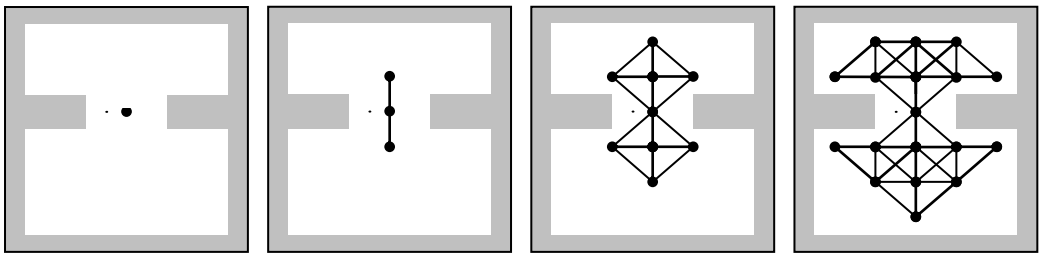


Figura 7: Algoritmo de preenchimento de grafos de navegação

Para realizar a busca por um item, pode-se utilizar qualquer algoritmo de busca em grafo. Porém existe uma diferença entre a busca de um caminho de um ponto a outro para a busca de um caminho para um item. Neste último caso, podem existir dezenas ou até centenas de itens do mesmo tipo no cenário. Deste modo, a estratégia de busca deve levar isso em consideração. Caso for utilizado o algoritmo A*, deve-se aplicar o algoritmo para cada item do cenário, o que é uma tarefa proibitiva. Uma solução é utilizar o algoritmo de Dijkstra, visto que ele procura radialmente em todas as direções. Tão logo ele encontrar um item, a busca é finalizada.

Outro exemplo onde o algoritmo A* não pode ser utilizado são jogos que incluem teletransportadores ou portais que o agente pode utilizar para se mover magicamente e instantaneamente entre locais. Isso ocorre porque não se tem como calcular a heurística h'. A localização do portal pode estar na direção oposta do destino, fazendo com que a busca nunca vá em sua direção. Para esses casos, deve-se utilizar o algoritmo de Dijkstra.

Grafos com granularidade maior dão uma maior cobertura do cenário, permitindo criar caminhos mais realistas. Porém, tendem a aumentar o custo de processamento. Isso é claramente observado em algoritmos para determinar o nó visível mais próximo a uma dada posição. Esse teste implica em calcular a visibilidade e distância em relação a todos os nós do grafo. Uma estratégia para amenizar este problema é utilizar alguma técnica de partição do espaço, como as técnicas de partição baseada em células, árvores BSP (*Binary space partition*), *quadtrees* , *navmesh*, ou algum outro método. Assim, a busca é dada em função da densidade do nó em questão, ao contrário do número de nós. Uma vez que geralmente o número de nós é igualmente distribuído, o tempo de busca tende a ser constante.

2.3. Caminho como nó ou aresta

O caminho gerado pelo algoritmo A* apresentado é composto por uma seqüência de vértices por onde o agente deve passar para chegar ao destino. Entretanto, geralmente caminhos definidos por arestas dão ao programador

uma flexibilidade adicional. Para isso vamos analisar um grafo de navegação que define os caminhos nas proximidades de um rio, como mostrado na Figura 8.

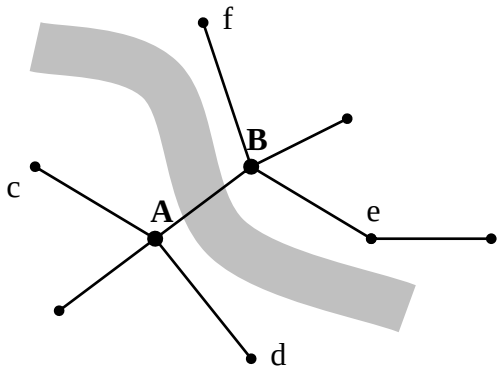


Figura 8: Grafo de navegação que atravessa um rio entre os nós A e B

Para que o agente atravesse o rio (aresta A–B), o game designer observa que é necessário que o agente mude seu comportamento para nadar, ao invés de caminhar. Para isso, os nós A e B possuem informação adicional para indicar essa mudança de comportamento (*observe que isso é um caso de colocar a IA no cenário e não no personagem*). Supondo que o agente está seguindo o caminho d-A-B-e. Quando o agente atinge o nó A, seu comportamento irá mudar para nadar, podendo assim chegar ao nó B. Quando ele atinge o nó B, este também tem informação associada ao comportamento nadar, de modo que o agente continuará nadando ao longo da aresta B-e. Outro caso ainda pior ocorre caso o personagem for do nó d ao nó c. Ao chegar no nó A ele começará a nadar, mesmo não tendo que atravessar o rio.

Este problema poderia ser facilmente resolvido se a informação de comportamento estivesse associada à aresta e não ao vértice. Desta forma, o agente pode consultar as arestas à medida que percorre o caminho, mudando seu comportamento sempre que necessário. Para o exemplo anterior, a única aresta com informação de nadar seria a aresta A-B, sendo as demais com informação de andar.

2.4. Suavização de Caminhos

Independente da forma de geração do caminho (vértice ou aresta), a solução encontrada por qualquer algoritmo de busca quando aplicado a um grafo do tipo grade, é uma linha que geralmente tem uma forma sinuosa (zig-zag), visto que o espaço é **discretizado** em pontos definidos pelos vértices da grade. O mesmo problema ocorre quanto o personagem precisa encontrar o ponto do grafo mais próximo, para então encontrar o caminho que o leve ao destino. Geralmente o personagem não está sobre um nó do grafo e por isso precisa percorrer um caminho que não se parece natural, como é apresentado ao logo desta seção. Uma solução para isso é fazer um pós-processamento no caminho encontrado. Geralmente sempre existe uma forma de suavizar o caminho encontrado, traçando-se um novo caminho que suaviza e remove vértices não necessários, produzindo desta forma caminhos mais realistas.

Uma estratégia simples para este problema é verificar a visibilidade entre arestas adjacentes. Se uma aresta das é supérfula, então as duas são substituídas por uma única, como mostrado na Figura 9.

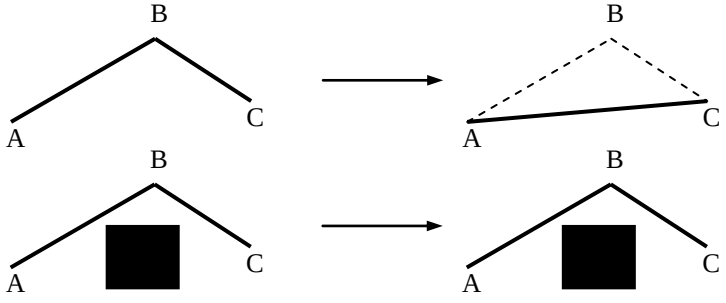


Figura 9: Situações para remoção de vértices de um caminho

O algoritmo opera do seguinte modo: Inicialmente dois iteradores de arestas A_1 e A_2 são posicionados na primeira e segunda aresta respectivamente. Então são realizados os seguintes passos:

1. Pega a posição origem de A_1;
2. Pega a posição destino de A_2;
3. Se o agente consegue se mover entre esses dois pontos sem nenhuma interseção com elementos estáticos do cenário, atribua a posição destino de A_1 para a de A_2 e remova a aresta A_2 do caminho. Reposicione A_2 na aresta seguinte a A_1 já modificada;
4. Se o agente não puder se mover entre esses dois pontos, atribua A_2 para A_1 e avance A_2 para a próxima aresta
5. Repita os passos até que o destino de A_2 seja igual ao destino final do caminho.

Na Figura 10 são ilustrados os passos do algoritmo, cujo caminho parte do quadrado e chega ao círculo.

Deve-se observar que este algoritmo é eficiente, porém não é ótimo. Como observado na Figura 10, as duas últimas arestas do caminho suavizado poderiam ser transformadas em uma única. O algoritmo não resolve isso pois somente testa arestas vizinhas. Dada uma aresta A_1, ela deveria ser testada com todas as demais cada vez que A_1 avança.

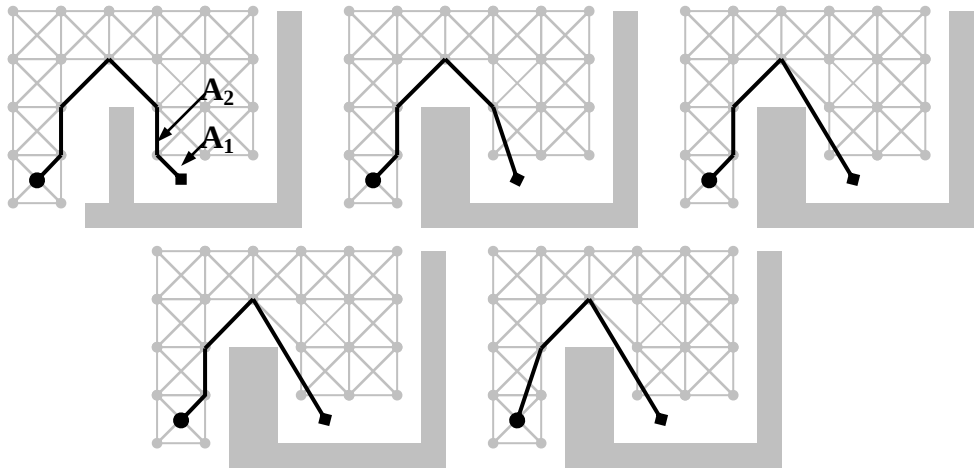


Figura 10: Etapas do algoritmo de suavização de caminhos

Exercício: Implemente o algoritmo de suavização para o caminho encontrado pelo algoritmo A*.

2.5. Técnicas para otimização de CPU

Pode ocorrer de em uma dada situação do jogo, várias unidades requererem a busca de caminhos simultaneamente. Isso pode gerar um pico de carga que pode resultar em interrupções momentâneas do fluxo do jogo. Obviamente, estas situações devem ser evitadas. Por isso, técnicas de otimização da busca de caminho devem ser aplicadas.

2.5.1. Caminhos pré-calculados

Se o cenário do jogo for estático, ou seja, se o custo de deslocamento entre nós nunca é alterado, uma estratégia para reduzir o uso da CPU é usar tabelas com valores pré-calculados, de modo que os caminhos de menor custo possam ser rapidamente encontrados. O pré-cálculo pode ser realizado em várias situações, como no momento que o jogo é carregado ou no momento que o game designer projeta o cenário. Neste último caso, juntamente com as informações do cenário, deve estar armazenada a tabela de busca. Essa tabela deve conter todas as rotas possíveis entre quaisquer dois nós. Para esse cálculo, pode-se utilizar o algoritmo de Dijkstra aplicado para criar a SPT a cada nó do grafo.

Para não armazenar todos os possíveis caminhos de cada nó a todos os outros, a tabela de busca faz uso de uma estratégia interessante. Ela armazena somente o próximo nó que deve ser seguindo do nó atual ao nó destino. Por exemplo, para determinar o caminho mais barato entre os nós C e E, toma-se o elemento da matriz que

corresponda a linha de origem e a coluna de destino. Neste caso, o nó escolhido é o nó B. Do nó B para o nó E, deve-se ir para D, e finalmente para E, complementando assim o caminho mais curto entre os dos pontos.

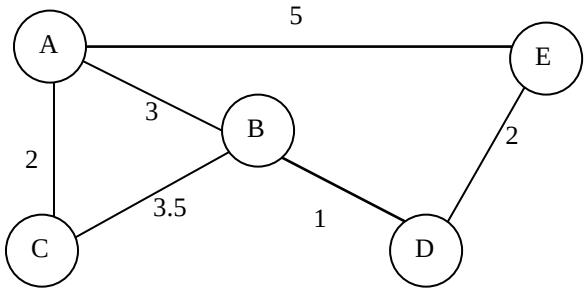


Figura 11: Grafo com pesos

	A	B	C	D	E
A	A	B	C	B	E
B	A	B	C	D	D
C	A	B	C	B	**B**
D	B	B	B	D	E
E	A	D	D	D	E

Figura 12: Tabela que representa os caminhos mais curtos para o grafo da Figura 11. Na figura está destacado o nó B, que é o primeiro nó que se deve ir para partir do nó C para chegar ao nó E.

Algoritmo:

1. Para cada nó origem do grafo
    a. Aplicar Dijkstra
    b. Calcular a SPT
    c. Para cada nó destino
        i. Se origem = destino
            1. path[origem][destino] = destino
        ii. Senão
        ... (faça como exercício)

Exercício: Implemente o algoritmo de pré-calculo de caminhos, a partir do algoritmo algoritmo A* (Dijkstra).

2.5.2. Custos pré-calculados

Saber qual o melhor caminho entre dois nós somente é útil quando se sabe onde se deseja ir. Porém em situações onde o agente precisa pegar um item, que pode estar presente em vários locais do mapa, não se sabe qual destes itens está mais próximo. Para isso, utiliza-se uma tabela com os custos de locomoção entre quaisquer dois nós, como mostrado na Figura 13. Uma vez sabendo-se qual o nó mais próximo, pode-se utilizar a tabela de caminhos para a rota mais curta ao destino selecionado.

	A	B	C	D	E
A	0	3	2	4	5
B	3	0	3.5	1	3
C	2	3.5	0	4.5	6.5
D	4	1	4.5	0	2
E	5	3	6.5	2	0

Figura 13: Custo para se deslocar entre quaisquer dois nós do grafo, para o grafo da Figura 11.

Algoritmo: Resolva em sala de aula

1. Para cada nó origem do grafo
    a. Aplicar Dijkstra
    b. Para cada nó destino do grafo
        i. Se origem <> destino
            1. custo[origem][destino] = calculaCusto();

Exercício: Implemente o algoritmo de pré-calculo de custos, a partir do algoritmo algoritmo A* (Dijkstra) apresentado na Seção 2.

2.5.3. Partilhamento do tempo (*Time-slicing*)

Uma outra forma de evitar os picos de processamento é alocar uma quantidade fixa de processamento da CPU para cálculos de caminhos a cada loop de processamento do jogo. Assim, todas as requisições de caminhos terão uma fração do tempo total alocado a este fim. Para que isso seja possível, devem-se fazer alterações nos algoritmos de busca, tanto para o A* como para o Dijkstra. A idéia é quebrar a busca em várias etapas. Isso é possível pois, analisado estes algoritmos, observa-se que a cada loop, um novo nó candidato é inserido na SPT, como mostrado resumidamente no seguinte algoritmo:

1. Selecione o melhor nó da Lista ABERTOS
2. Adicione o nó a SPT
3. Teste se o nó é o destino
4. Se não for o destino, gere a lista de SUCESSORES
5. Avalie os nós de SUCESSORES a adicione a ABERTOS

Para executar a busca em etapas, cada busca deve estar associada a um objeto que mantém o estado corrente de busca. Todas as buscas ativas são controladas por um gerenciador, que dado um tempo estipulado, pode determinar quantos ciclos cada busca pode realizar a cada loop do jogo. Quando uma busca termina, tanto em caso de sucesso ou fracasso, o gerenciador comunica o proprietário da busca por meio de uma mensagem.

Com esta estratégia, pode-se garantir um tempo constante no processo de busca a cada passo, independente da quantidade de agentes requisitando caminhos. Porém, deve-se observar que quanto mais agentes solicitarem buscas, maior vai ser o tempo (loop do jogo) até que todos os caminhos sejam encontrados.

Com esta estratégia, pode haver um *delay* do momento que o agente solicita um caminho até o momento que recebe uma notificação que o caminho foi encontrado. Este delay é proporcional ao tamanho do grafo de navegação, ao número de ciclos por cada atualização alocada ao gerenciador, e pelo número de pesquisas ativas. Em jogos onde o jogador designa uma ação de deslocamento a um personagem, é esperado que o agente responda imediatamente. Porém, se o caminho não foi calculado, uma solução é fazer com que o agente comece a andar na direção do destino, enquanto o caminho é processado. Caso o destino não for conhecido, como ocorre no caso de busca de itens, a solução é fazer o personagem fique vagando até receber a notificação. Para ambos os casos, tão logo o agente receba o caminho selecionado, geralmente ele não se encontra mais na posição original, e por isso o caminho pode não ser mais válido. Uma solução é fazer com que o agente volte à posição original e siga o caminho estipulado. Uma solução mais apropriada vem como resultado do algoritmo de suavização de caminhos apresentado anteriormente. Após a aplicação deste algoritmo, os nós (*waypoints*) não mais necessários são removidos do caminho, como mostrado na Figura 14.

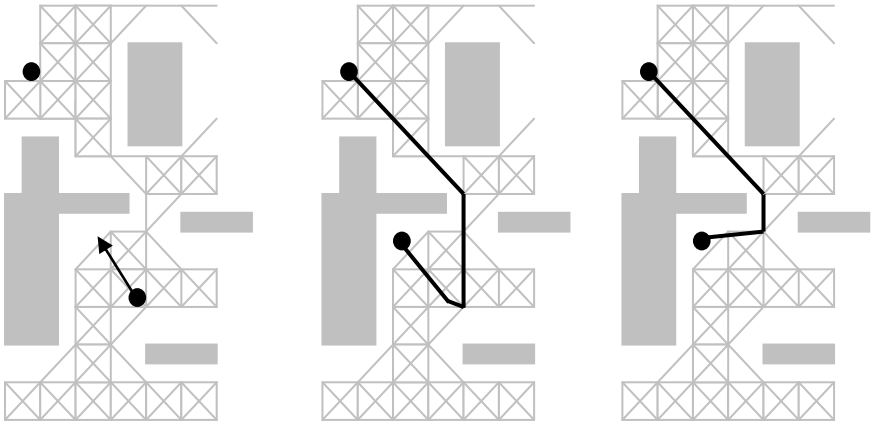


Figura 14: Geração de caminho baseado em uma estratégia de *time-slicing*

2.5.4. Path-planning hierárquico

Outra técnica para gerenciar o uso excessivo de CPU com busca de caminhos é chamada de *hierarchical pathfinding* (busca de caminho em hierarquia). Como o próprio nome diz, esta estratégia consiste em achar o caminho de modo gradativo, partindo-se de uma solução grosseira até uma solução mais refinada. Fazendo-se uma analogia com situações da vida cotidiana, vamos considerar os passos necessários para se locomover dentro de uma casa. Supondo que se esteja no quarto e se deseja ir até a cozinha, pode-se pensar no caminho com uma

seqüência de partes da casa que terão que ser percorridas, como ir ao corredor, descer as escadas, passar pela sala para chegar a cozinha. Em outro nível de detalhe, pode-se definir o caminho a ser seguido dentro de cada peça da casa, ou seja, como atravessar o quarto, a sala e como chegar na geladeira dentro da cozinho. Em outras palavras, o caminho pode ser planejado usando uma série de áreas (salas, corredores, etc) e em outro nível mais baixo uma série de objetos e pontos (cadeiras, tapetes, etc).

Da mesma forma como neste exemplo, a busca de caminho em jogos pode ter vários níveis de granularidade (refinamento). Como exemplo, suponha que se deseje fazer um jogo que retrata uma guerra nacional (no Brasil). Em um nível mais alto, pode-se considerar cada estado como um nó do grafo, e um nível mais refinado, cada cidade e estrada como nós. Quando uma unidade requer um caminho entre Porto Alegre e São Paulo, o seguinte caminho seria gerado: Rio Grande do Sul → Santa Catarina → Paraná → São Paulo. Este é um processamento extremamente rápido, visto que existem poucos nós neste grafo. A partir deste caminho simplificado, pode-se então gerar um caminho mais detalhado que mostra as estradas que devem ser seguidas. Geralmente o número de níveis em jogos é 2, porém nada impede que se utilize mais de 2 níveis. Por exemplo, no primeiro nível pode-se representar os estados, no segundo nível as cidades e estradas e no terceiro nível as ruas dentro das cidades. Na Figura 15 são apresentados exemplos de mapas em dois níveis de granularidade.

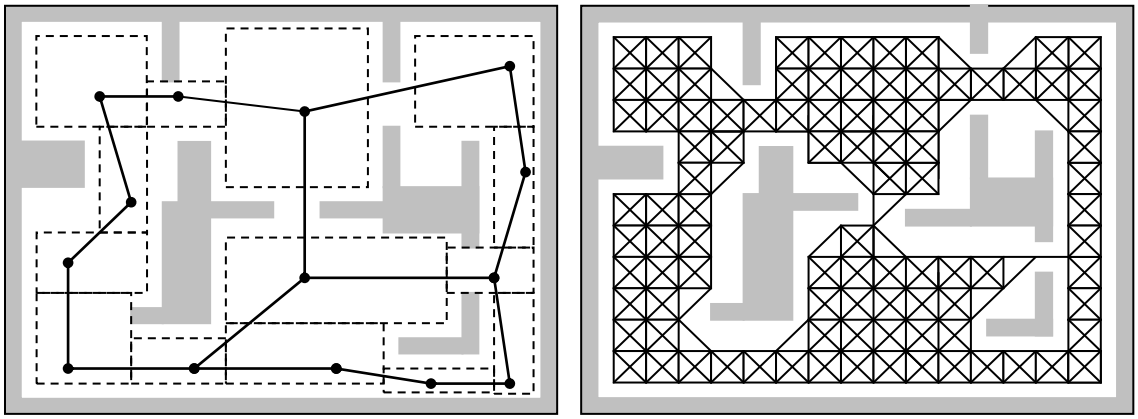


Figura 15: Planejamento de caminho hierárquico

Referências

[1] Antonio C. Mariani**. Conceitos Básicos da Teoria de Grafos**. Disponível em: http://www.inf.ufsc.br/grafos/definicoes/definicao.html

[2] Matt Buckland. **Programming Game AI by Example**. Wordware publishing Inc, 2005 (*Referência usada na maior parte deste material*)

[3] Matt Buckland. **Programming Game AI by Example**. Resource Page. Disponível em: http://www.wordware.com/files/ai/

[4] Rich, E., Knight, K. **Inteligência Artificial**. Makron Books, 1994.