Inteligência Artificial para Jogos

Aula 08 - Descoberta de caminho - Parte 02

Apresentação da Aula

Olá! Seja bem-vindo(a) à aula 08! Na aula passada, você conheceu um algoritmo que permitia traçar uma linha reta em um espaço discreto. Embora seja um algoritmo interessante, ele apenas serviu como um primeiro passo para lhe mostrar que mesmo uma tarefa simples como andar em linha reta pode exigir um algoritmo elaborado para o computador entender.

E o que acontece quando se precisa traçar caminhos mais complexos, que desviam de obstáculos, como o apresentado na aula passada? Hum... pensou? Isso mesmo! Você vai precisar de algoritmos ainda mais elaborados!

Antes de mais nada, você precisará entender mais uma forma de organizar o espaço do jogo de maneira que permita a implementação desses algoritmos. Para isso, você irá aprender, de forma bem intuitiva, uma das estruturas de dados mais importantes da computação: os **grafos!**

Preparado pra começar?

Objetivos

> Conhecer a estrutura de dados **Grafo**;
>
> Compreender como problemas podem ser modelados com essa estrutura;
>
> Aprender formas de implementar um **Grafo** computacionalmente e as diferenças entre as abordagens.

1 - Representando o espaço do jogo como um grafo

Na aula passada, você conheceu o **algoritmo de Bresenham**, que é ótimo para traçar retas (ou algo bem próximo de uma reta! 😁) em espaços discretos. O problema é que nem sempre o caminho direto até o objetivo está livre! Às vezes tem um Snorlax dormindo por ali, e você vai ter de pensar em um outro jeito de chegar no próximo ginásio... 😉

Figura 01 - Bloqueio “natural” do caminho

Quando você se encontra em um cenário em que é necessário traçar um caminho que desvie de obstáculos ou áreas onde não se pode caminhar (ninguém quer brincar de andar pela lava não é mesmo?), torna-se necessário usar algoritmos mais sofisticados, que entendam que andar por um terreno de grama macia é bem melhor do que andar pelo terreno de pedras pontiagudas. Você precisará de algoritmos que entendem a noção de que um caminho é melhor do que o outro!

Para compreender esses algoritmos, você precisará aprender uma estrutura de dados que facilitará em muito a sua vida: os **grafos**.

Grafos são uma teoria matemática extremamente rica e complexa, com muitos detalhes, formalismos e teoremas a provar, mas não irei abordar esse lado matemático aqui! O que vai se precisar por enquanto é entender o conceito do **grafo** e como se pode implementar um no computador, ok?

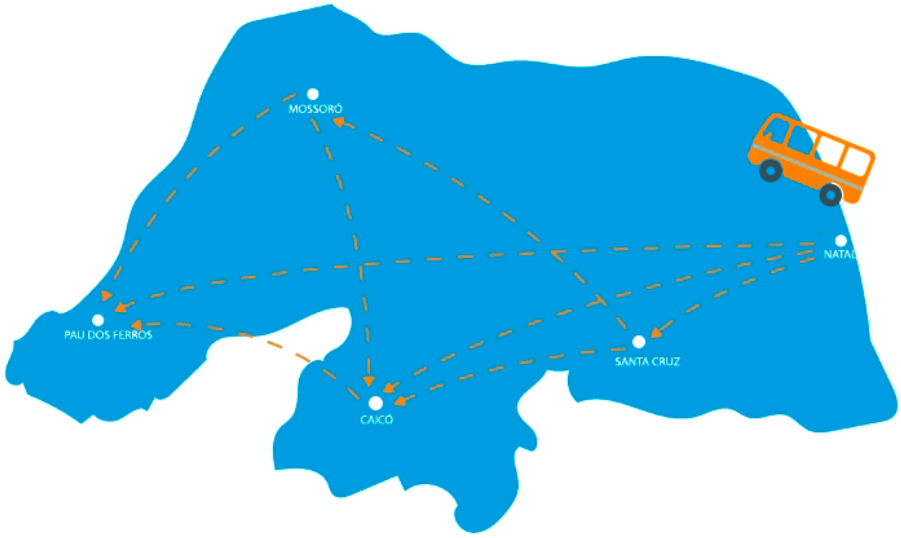
Em primeiro lugar, o **grafo** é uma estrutura que representa as informações em função de dois elementos: **vértices** (ou nós) e **arestas**. Os **vértices** normalmente representam um dado ou uma informação, enquanto as **arestas** representam uma ligação entre esses dados.

Ok, ficou um pouco vago não foi? Dando um exemplo então: imagine que você precisa programar uma viagem pelo estado, e os preços da gasolina, mais infinitos do que a guerra dos Vingadores, estão fazendo as passagens de ônibus ficarem muito caras.

Fonte: Adaptado de COUCH TOMATO e JORNADA GEEK. Disponível em: http://couchtomatonews.com/where-are-they-now-dr-bruce-banner/ e https://www.jornadageek.com.br/novidades/vingadores-guerra-infinta-ruffalo-banner-hulk/. Acesso em: 24 maio 2018.

Então para começar seu planejamento, você faz uma lista das cidades do estado (não esqueça das que você quer visitar) e do preço das passagens entre elas. Pronto! Imagino que você chegou em algo como o que é visto na Figura 02.

Figura 02 - Planejando sua viagem!

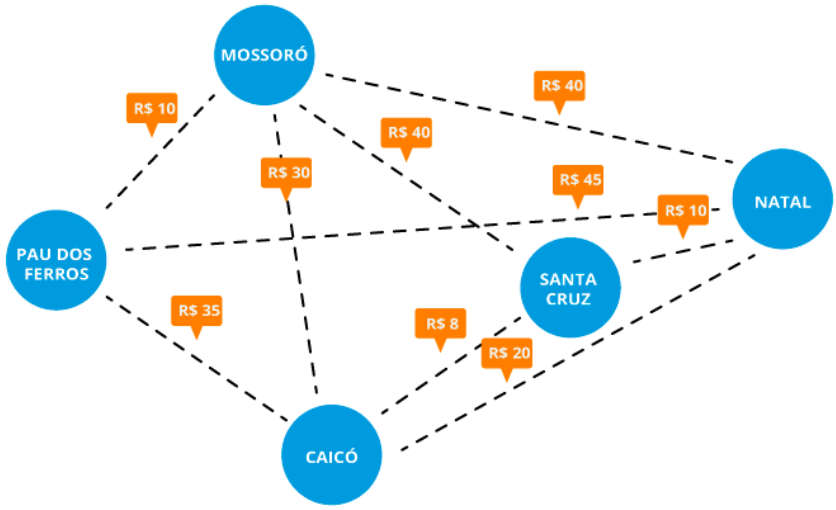


Que mapa bonito, hein? 😁 Perceba que você pode fazer várias análises em cima desse mapa:

- Não existe uma linha direta de Santa Cruz para Pau dos Ferros, quem quiser viajar de um para o outro vai ter de ir primeiro para outra cidade!
- Viajar direto de Natal para Pau dos Ferros (45 reais) é mais barato do que viajar para Mossoró (40) e depois ir para Pau dos Ferros (10);
- Viajar de Natal para Caicó (20 reais) é mais caro do que ir primeiro para Santa Cruz (10) e de lá para Caicó (8).

Agora eu vou contar um segredo... 🤫 o mapa acima pode ser representado por um **grafo!** As cidades seriam os vértices, enquanto as retas (que representam as rotas da viagem) seriam as arestas. Mas a aresta especifica mais do que uma ligação entre as cidades: ela também traz o quanto custa ir de uma cidade para a outra. Quando isso ocorre, se diz então que essa aresta tem um **peso** ou **custo** de travessia. Existem **grafos** em que as arestas não têm peso, e normalmente só representam a ligação entre os vértices. Já já explico sobre isso! Primeiro vou colocar aqui um desenho mais clássico do **grafo** (Figura 03), que representa o mapa (Figura 02) acima:

Figura 03 - **Grafo** das cidades



Um outro conceito que vem dessa estrutura de **grafo** é a noção de **caminho**, que nada mais é do que o percurso que você faz de um nó a outro no grafo. Por exemplo, um caminho de Natal para Santa Cruz pode ser construído de várias formas:

- Direto, Natal para Santa Cruz;

- Passando por Caicó, depois Santa Cruz;

- Passando por Mossoró, depois Santa Cruz;

- Passando por Mossoró, depois Caicó, depois Santa Cruz;

- Passando por Pau dos Ferros, Mossoró, Babilônia, Japão...

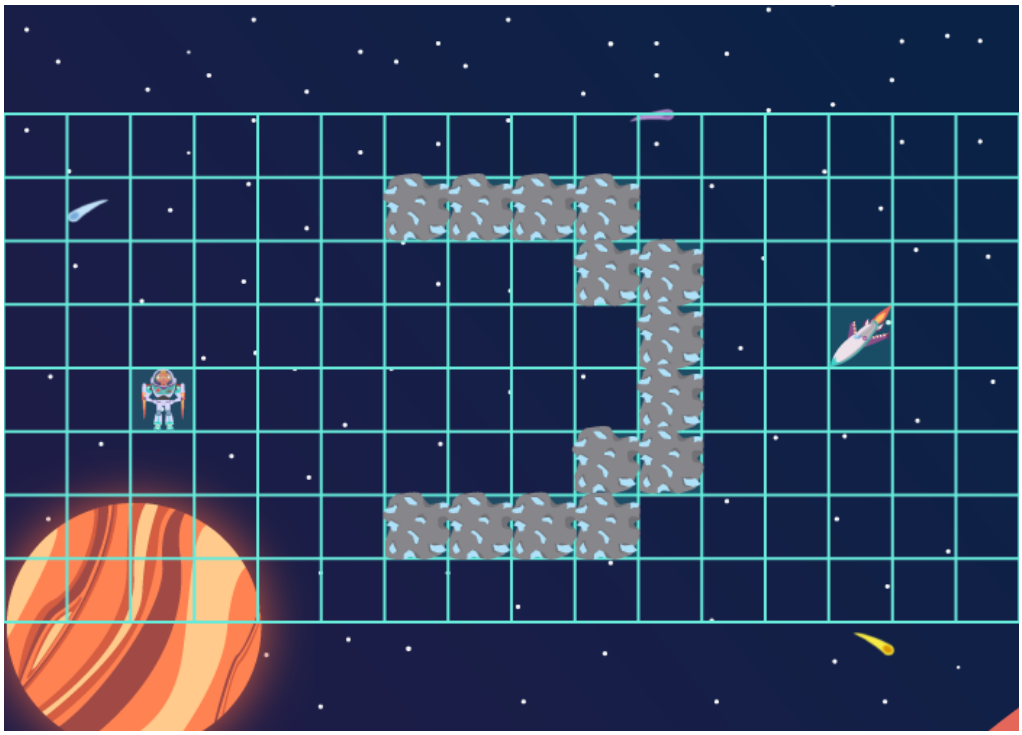
Deu pra entender? Um **caminho** é composto pela sequência de vértices que você visita de um ponto A para um ponto B, independentemente se é uma sequência mais direta até o objetivo, ou se você vai arrodear e gastar todo o dinheiro com passagens! Na verdade, a lista acima representa vários caminhos possíveis entre Natal e Santa Cruz (ok, o último não, porque coloquei cidades que não estavam no grafo 😁). E qualquer caminho desses serve, não é?

Depende. Nesse caso, seria melhor escolher um caminho que gastasse menos dinheiro, não é mesmo? Então o caminho direto, que tem um custo de 10 reais, é melhor do que os outros caminhos, que vão para mais de 30 reais. Quando existe um caminho entre dois pontos cujo custo para percorrer é menor do que os outros, se diz que ele é o **menor caminho** entre os dois pontos. Se um **grafo** não possui peso nas arestas, essa noção de **menor caminho** é diretamente associada com o tamanho do caminho (afinal é tudo igual, então quanto menos arestas percorrer, menor será o caminho!). O mesmo não é verdade quando se tem um **grafo** com peso nas arestas: o **menor caminho** pode passar por mais vértices e arestas do que o caminho direto.

Veja por exemplo o caminho entre Natal e Caicó: o caminho direto tem um custo de 20 reais, enquanto o caminho que passa por Santa Cruz custa 10 reais para chegar em Santa Cruz e 8 reais para chegar em Caicó, totalizando 18 reais (dá pra comprar um suco e um salgado na rodoviária!). Dessa forma, o **menor caminho** nesse caso passa por mais vértices do que o caminho direto.

Porque quando se trabalha com cenários de jogos, principalmente com obstáculos e terrenos, você irá utilizar essa noção! Lembra dessa figura da aula passada?

Figura 04 - Olha eu de novo!



Pode-se representar esse mapa como um **grafo** também, seguindo a mesma ideia da aula passada: cada célula (quadrado) do mapa será um vértice do **grafo**. E as arestas? Serão o custo de movimentação entre os quadrados. Por padrão, ir de um quadrado para o outro custará 1 (a mesma ideia da aula passada).

Mas se você for fazer tudo igual a aula passada, ele não vai traçar uma linha reta do mesmo jeito e ficar preso?

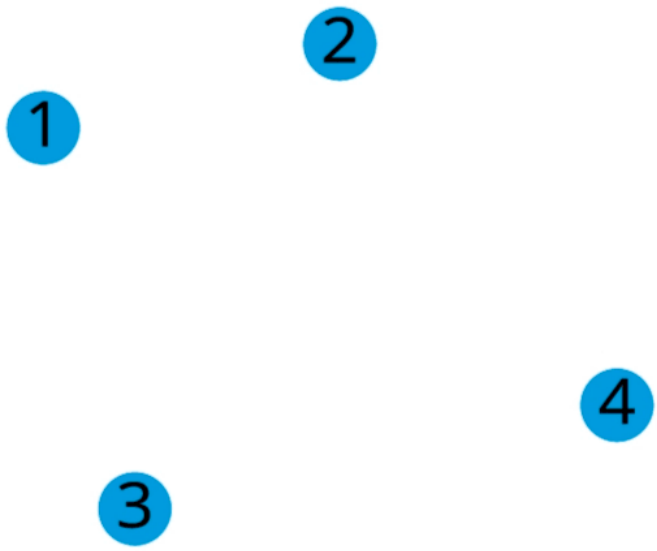
Se tudo tiver 1 como custo, sim! Mas uma estratégia que se pode fazer é atribuir um custo muito grande para os quadrados que contêm as pedras e os que estão em volta deles. Por exemplo, imagine que os quadrados com pedras teriam um custo de um milhão para serem atravessados! Dessa forma, a tendência é que o caminho passe por casas que fazem a volta pelo lado de fora do obstáculo, desviando e permitindo que o personagem chegue ao seu objetivo, que é a nave.

Essa estratégia também pode ser usada em jogos que possuem terrenos distintos. Nesse caso, não há um bloqueio do movimento do personagem, apenas um custo adicional para transpor determinados tipos de terreno (vegetação densa, pântanos, desertos). É comum que uma rota traçada pelo computador leve o personagem por um caminho onde ele possa se movimentar um maior número de

casas conservando o máximo de energia, o que significa preferir caminhos em terrenos mais suaves, indo para os terrenos mais árduos apenas quando não há mais opção.

Uma coisa que vale a pena destacar: no **grafo** do exemplo, estou assumindo que o custo de ir de Natal para Mossoró é igual ao custo de ir de Mossoró para Natal. Ou seja, o **grafo** é **não-direcionado**: a aresta serve como mão-dupla, e nosso caminho pode ir e voltar por ela com o mesmo valor. Existem **grafos** onde as arestas são **direcionadas**, ou seja, é uma via de uma mão: você pode ir, mas não pode voltar. Nesses casos pode ocorrer de ter um custo para ir do nó A até o nó B, e um custo diferente para ir do nó B até o nó A. Na representação visual, as arestas possuem uma seta indicando a direção do movimento.

Figura 05 - Exemplo de **Grafo** direcionado



Você agora vai ver como se pode implementar os **grafos** computacionalmente!

2 - Como representar grafos no computador

Existem duas abordagens clássicas para implementar um **grafo** para uso nos algoritmos: como uma **matriz de adjacência** (ou vizinhança) ou uma **lista de adjacência** (ou vizinhos). Você vai ver as duas formas e em que momento seria mais vantajoso usar cada uma delas. Os critérios utilizados para diferenciar as formas de representação serão memória necessária para armazenar o **grafo** *versus* velocidade de acesso à informação.

Conheça agora os candidatos! 😉

2.1 - Matriz de Adjacência

Essa é, para mim, a forma mais intuitiva de representar um **grafo** computacionalmente. A ideia é montar uma grande matriz, onde as linhas e colunas representarão os vértices do **grafo**, como se estivesse fazendo um cruzamento de todos os vértices. Nesse caso, cada posição da matriz representará uma aresta que liga o elemento da linha X com o elemento da coluna Y.

Veja um exemplo ilustrado no Quadro 01 para entender melhor. Abaixo segue a matriz de adjacência para o **grafo** do RN, apresentado no começo da aula:

Quadro 01 - Matriz de adjacência (**Grafo** do RN)

Cidades	**Natal**	**Mossoró**	**Santa Cruz**	**Caicó**	**Pau dos Ferros**
Natal	0	40	10	20	45
Mossoró	40	0	40	30	10
Santa Cruz	10	40	0	8	-1
Caicó	20	30	8	0	35
Pau dos Ferros	45	10	-1	35	0

Dessa forma, se quiser consultar qual o custo de ir de Santa Cruz até Mossoró (diretamente), basta consultar o Quadro 01 acima! Alguns detalhes: o custo de ir de uma cidade para ela mesma é zero (não gasta nada). Estou descartando qualquer custo de deslocamento interno no **grafo**. 😉 Outra coisa interessante na quadro é o valor de -1 que apareceu entre Pau dos Ferros e Santa Cruz. Isso acontece porque não existe ligação direta entre as duas cidades, e é impossível ir de uma para outra sem passar por outra cidade primeiro! Como -1 é um valor impossível em reais (claramente, o pessoal que inventou grafos nunca viu minha conta do banco 🤭), coloquei para representar um peso infinito! Sacaram a estratégia?

As principais vantagens dessa representação é que as linguagens de programação normalmente possuem um tipo de matriz disponível para implementação, e o acesso ao valor de qualquer aresta é muito rápido (basta olhar a posição da matriz na linha e coluna desejada!).

Então se é tão bom, não precisava ter inventado outras formas de armazenar um **grafo** né?

Precisava... 😢

Infelizmente, a representação por matriz é desvantajosa em algumas situações:

- **Grafos** com um número de nós muito grande vão exigir uma matriz enorme para representar. Imagina o gráfico de relações no Facebook para ele ficar lhe sugerindo amiguinhos: seria necessária uma grande quantidade de memória para armazenar a matriz, e seu programa ia começar a precisar de alguns pentes extra de memória RAM para rodar;

- **Grafos com muitos vértices e poucas ligações**, denominados **grafos esparsos**, teriam muitas casas da matriz armazenando um valor que indica que não há ligação (no nosso exemplo, o -1). Nesse caso, ocorreria muito desperdício de memória para representar poucos dados!

Então essa é uma ótima opção quando:

- O conjunto de vértices é pequeno ou médio;

- O **grafo** é bem preenchido (muitas ligações entre os nós).

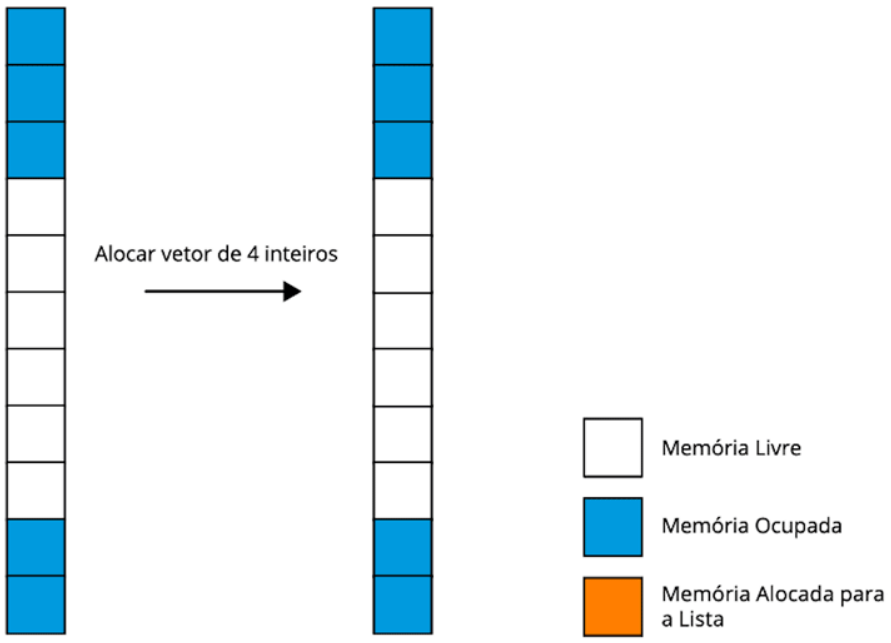
E quando não se tem um **grafo** desses em mãos? Veja a segunda opção!

2.2 - Lista de Adjacência

A segunda forma de representação utiliza uma estrutura de dados clássica da computação para montar o **grafo**: uma **lista** ou **lista ligada**. Mas o que danado é isso?

Lembra que lá em programação estruturada você estudou os arranjos, especificamente vetores e matrizes (se não lembra, pelo amor de Deus dê uma olhadinha lá!)? Então, essas estruturas de dados são ditas **sequenciais** porque elas são armazenadas de forma sequencial. O que isso significa é que o computador vai deixar tudo juntinho na memória! Veja a Figura 06.

Figura 06 - Exemplo de alocação de vetor na memória

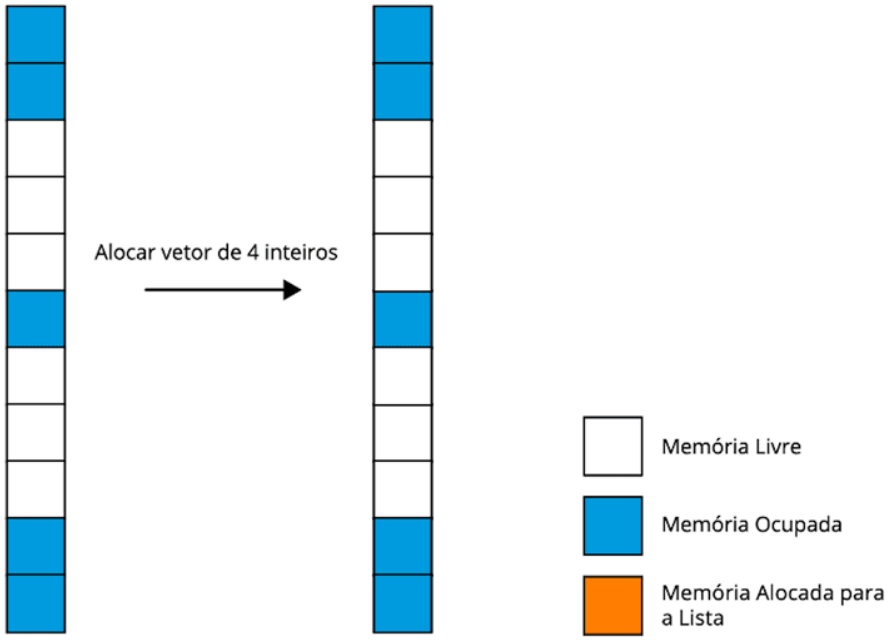


Foi como um exemplo visual para refrescar bem a sua memória!

Fonte: TENOR. Disponível em: https://tenor.com/. Acesso em: 24 maio 2018.

Assuma que cada posição de memória possui 4 bytes (o suficiente para armazenar um valor do tipo inteiro). Como existiam quatro posições consecutivas livres, o vetor foi alocado com sucesso. Agora se o cenário fosse o seguinte:

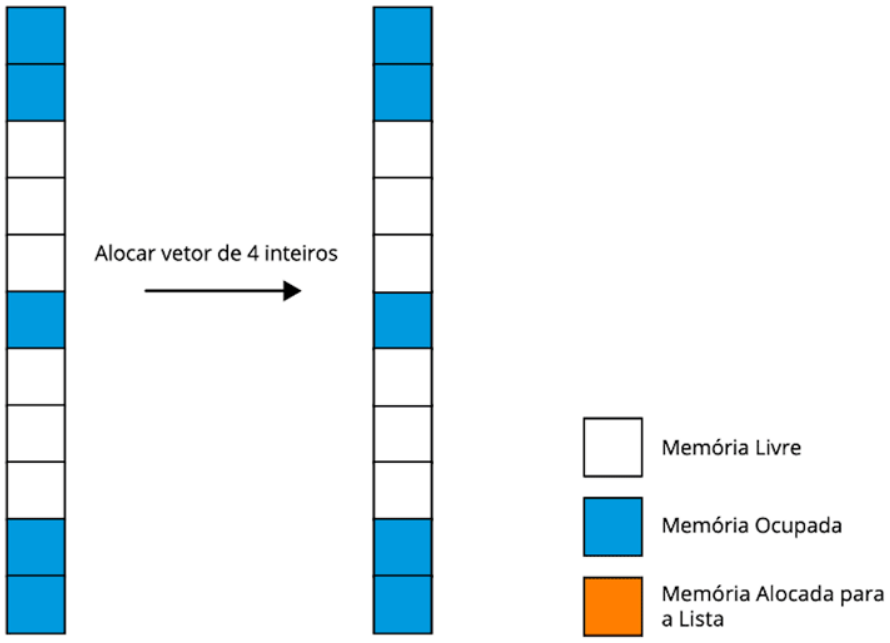
Figura 07 - Exemplo de alocação malsucedida



Opa! Mesmo que a quantidade de memória livre seja a mesma (6 espaços nos dois cenários), no segundo não existe a quantidade necessária de casas em sequência para armazenar o vetor, logo o seu programa não vai conseguir executar a operação (provavelmente vai vir aquele ***Segmentation Fault*** (mensagem de erro de memória na linguagem C) ou ***Null Pointer Exception*** (mensagem de erro de memória na linguagem Java)).

É normal que, com o tempo de uso, abrindo e fechando programas, e executando milhões de operações, a memória do seu computador comece a ficar ***fragmentada*** (termo para quando a ocupação da memória começa a ficar com pequenos espaços não utilizados). E você viu lá em Arquitetura de Computadores e SO: quando acaba a memória, vem a danada da memória virtual (que usa o disco rígido), cai o desempenho da máquina e os usuários choram de dor e xingam nossos familiares... Para evitar que isso ocorra, foram idealizadas estruturas de dados que armazenam o dado de forma não-sequencial. Basicamente, elas guardam duas informações: o próprio valor que se quer armazenar, e onde o próximo valor da sequência está guardado na memória!

Figura 08 - Listas resolvem seus problemas!



Professor, isso parece que dá trabalho de fazer!

Relaxe! Hoje em dia quase todas as linguagens possuem um tipo lista embutidas, ou uma implementação já pronta que você pode usar. Assim como as pilhas e filas, as listas podem ser utilizadas pelo elemento List do C#, então você só precisa entender como ela funciona, ok?

O resumo da ópera é: uma lista parece com um vetor, só que ela é armazenada de forma não-sequencial na memória. Qual a diferença então? É só performance: como a lista tem de descobrir a cada elemento onde está o próximo dela na memória, ela demora um pouquinho mais para executar do que o vetor. Mas essa diferença só vai ser sentida se for uma sequência muito grande de operações, então você pode utilizá-la a vontade para os exemplos daqui!

Agora que você tem uma noção do que é uma lista, vou lhe explicar o que é uma lista de adjacência! Uma lista de adjacência é... um vetor contendo todos vértices do **grafo**, onde cada posição será uma lista com os vizinhos daquele vértice! Observe a Figura 09 para clarear um pouco a explicação.

Figura 09 - Lista de Adjacência do grafo do RN

Nesse caso, o que você vai ter é um vetor de listas, onde cada lista contém apenas os vértices e o valor da aresta que o liga ao elemento correspondente do vetor. Essa abordagem é ótima para **grafos** esparsos ou com uma grande quantidade de vértices porque você só vai criar o que realmente existe, porém o tempo de acesso é mais demorado: para cada vértice inicial da aresta você vai ter de usar um laço de repetição para percorrer a lista até achar o nó de destino que você deseja (caso ele exista! Se não tiver uma ligação, você vai varrer a lista toda até descobrir isso 😉).

Como você pode ver, são estratégias de armazenamento de **grafos** para situações complementares, então dependendo da estrutura do seu **grafo**, você pode escolher aquela que será mais adequada para representá-lo no computador. 😁

Até aqui foram apresentados o que é um **grafo** e como se pode representá-lo em um algoritmo de computador. Eu sei que ficou uma aula super teórica, mas nas próximas duas aulas você vai ver vários algoritmos que fazem percursos e caminhos

em cima dessa estrutura. Então dê uma boa revisada e,, caso tenha dúvidas, não hesite de atacar os fóruns com uma chuva de perguntas.

Resumo

Nesta aula foi abordado, de forma intuitiva, o conceito de grafos no aspecto computacional. Essa estrutura de dados é uma das (se não a) mais importantes da computação, e além das aplicações dentro da área de jogos, são usadas para resolver problemas em diversas áreas da Ciência da Computação e Engenharia de Software. Facebook, Google, Microsoft e Petrobras são exemplos de empresas em que os **grafos** são a base para a construção das soluções do dia-a-dia!

Nesse contexto, os **grafos** foram apresentados como uma forma de representar o espaço do mapa do jogo, dando valores para passagem entre as células do mapa. A ideia é, a partir desse ponto, construir caminhos que permitam aos personagens desviar de obstáculos com antecedência, e não ficarem presos no meio do cenário. Essas técnicas são bastante utilizadas em jogos de RTS e MOBA para traçar o caminho das unidades (menos quando o jogador fica utilizando a técnica de mil cliques infinitos por segundo, aí não tem computador que calcule rota a tempo 
).

Foram abordadas também duas alternativas, em termos de estruturas de dados computacionais, para se implementar um **grafo** no computador. Para cada uma delas, foram ressaltadas as vantagens e desvantagens.

Na próxima aula você poderá ver essas implementações em ação, quando forem introduzidos os primeiros algoritmos de busca de caminhos em **grafos**.

Até lá!

Referências

CORMEN, Thomas H. et al. **Introduction to algorithms third edition**. Massachusetts: MIT Press, 2009.

KHAN ACADEMY, **Representando GRAFOS**. Disponível em https://pt.khanacademy.org/computing/computer-science/algorithms/graph-representation/a/representing-graphs. Acesso em: 21 maio 2018.

KYAW, Aung Sithu. **Unity 4. x game AI programming**. Birmingham: Packt Publishing Ltd, 2013.