Inteligência Artificial para Jogos

Aula 09 - Descoberta de caminho – Parte 03

Apresentação da Aula

Olá! Na aula de hoje você verá os primeiros algoritmos que traçam um caminho dentro de um **grafo** e que já poderiam ser usados dentro dos seus jogos para realizar diversas funções, desde traçar um caminho de um personagem de um ponto X até um ponto Y, até testar se um mapa que foi gerado proceduralmente pode ser percorrido por todos os pontos sem que o personagem fique preso!

Todos os algoritmos apresentados aqui têm como base a estrutura de **grafos** que foi apresentada na aula passada, por isso não hesite em voltar um pouquinho para revisar o assunto caso não lembre de um conceito ou outro, ok?

Primeiro, você verá a ideia por trás do algoritmo, e só depois como seria o código dele, de forma que você possa entender bem como ele funciona e a motivação para o seu desenvolvimento. Esses algoritmos são construídos não apenas em termo da função que eles exercem, mas pensando também no desempenho computacional no momento da execução.

Preparado? Então prossiga!

Objetivos

> Conhecer os algoritmos de **Busca em Profundidade** (DFS) e **Busca em Largura** (BFS);
>
> Observar o funcionamento dos algoritmos a partir de exemplos;
>
> Aprender o código de implementação dos algoritmos em C#.

1 - Caminhos em um grafo

Você já conheceu a estrutura de **grafos** e viu alguns exemplos de como modelar problemas através dela, além de algumas opções de como implementá-los computacionalmente. E caminhar pelo bichinho que é bom, nada!

Mas agora você terá a oportunidade, pois chegou a hora de apresentar algumas formas de traçar caminhos em **grafos.** 😉

Inicialmente, não se preocupe em achar o menor caminho (ainda): foque o estudo em alguns algoritmos que permitem realizar passeios pelo **grafo**, encontrando um caminho (ou todos) de um ponto A até um ponto B da estrutura.

Existem dois algoritmos clássicos de busca em **grafos**, denominados **Busca em Profundidade** (*Depth-First Search*) e **Busca em Largura** (*Breadth-First Search*), carinhosamente conhecidos por suas siglas, **DFS** e **BFS**, respectivamente.

Sim, são até parecidas, para facilitar a confusão. 😁

Figura 01 - Equipe Busca em **Grafos** decolando na velocidade da luz?

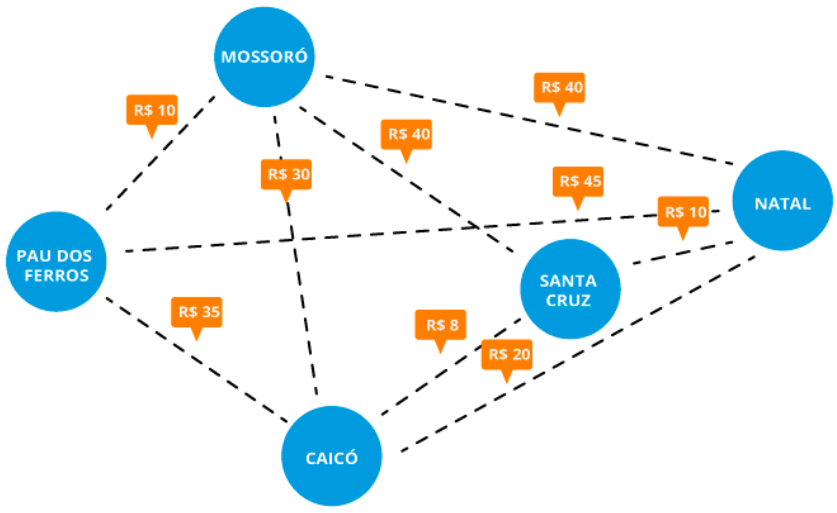
Fonte: AMINO. Disponível em: <https://aminoapps.com/c/pokemon/page/item/team-iaosket/DzhN_IX2BjZV6eeWX6oWNjPqMerrw>. Acesso em: 04 jun. 2018.

A ideia por trás dos dois algoritmos é similar: você vai escolher um vértice inicial do **grafo**, o que você quiser, e, a partir dele, vai tentar chegar naqueles vértices que têm uma ligação direta (ou seja, uma aresta) com ele. Nesse momento você coloca esses vértices no caminho e repete a operação até chegar no ponto final que você quer. Sim, não ache estranho, a ideia é essa mesmo! De ponto em ponto, até chegar no final! 😉

Antes de eu sair mostrando códigos, quero frisar que, apesar de seguirem a mesma ideia, existe uma diferença básica entre os dois algoritmos: a **ordem** em que eles percorrem os vértices vizinhos. Também se fosse tudo igual, ia ser um algoritmo só, não é mesmo?

Então utilize recursos visuais para ajudar. E volte ao lindo **grafo** do RN, que foi visto na aula passada.

Figura 02 - **Grafo** base para a aula



2 - Busca em Profundidade

No algoritmo **DFS**, percorre-se o **grafo** em termos de profundidade. Essa noção relaciona-se ao quão longe do vértice de origem o caminho vai parar dentro do **grafo**: a quantas arestas do vértice original o vértice atual do caminho se encontra? Se o caminho passa por uma aresta, a profundidade é de um. Se passa por três arestas, a profundidade é de três. O nome **busca em profundidade** advém de o algoritmo estar sempre tentando ir o mais “profundo” (distante) possível do vértice origem na sua busca pelo vértice destino. Também ocorre uma grande variação da profundidade ao longo da execução, uma vez que, ao não encontrar mais caminho para seguir em frente, o algoritmo retorna para o vértice anterior e tenta achar uma nova rota a partir dele, explorando vizinhos que ainda não foram visitados.

Figura 03 - Não tão profundo...

Calma, vou explicar um pouco melhor. No **grafo** do estado do RN, imagine que você quer fazer um caminho entre Santa Cruz e Pau dos Ferros. O vértice inicial da viagem é Santa Cruz, mas quem são os vizinhos dele?

Ora, os vizinhos são todos os vértices que têm uma aresta conectando com Santa Cruz, nesse caso, Natal, Mossoró e Caicó. Ao procurar um caminho para Pau dos Ferros, você poderia começar por qualquer um desses vizinhos. Então escolha Caicó, pra poder comer uma bolachinha na manteiga! 

Em uma **busca em profundidade**, os outros vizinhos de Santa Cruz não serão explorados nesse momento: primeiramente serão explorados os vizinhos de Caicó, que estão em uma profundidade maior no caminho (em Santa Cruz você deu zero passos, em Caicó, já deu um passo). Faça isso sucessivamente (explore o vértice em dois, três, mil passos!) até chegar no destino. Depois que terminar de explorar, pode voltar para os vértices em níveis anteriores e continuar explorando outros caminhos e vizinhos que ainda não foram verificados. Captou a mensagem?

Observe o pseudocódigo do algoritmo abaixo (Código 01), talvez facilite um pouco a sua percepção de como isso ocorre:

Código 01 – Pseudocódigo da **DFS**

```
Função DFS(int atual, int destino){
   Adiciona o vértice atual ao caminho;
   Marca como vértice como visitado;

   Se atual == destino
      imprime/retorna o caminho construído
   Senão{
      Para cada um dos vizinhos não visitados de atual{
         DFS(vizinho, destino)
         //Se você quiser todos os caminhos
         //Desmarca o vizinho como visitado
         //Retira o vizinho do caminho
      }
   }
}
```

Ainda não, né? Então veja as ilustrações abaixo! Os vértices verdes representam o ponto atual do caminho, os vértices em vermelho os pontos já visitados e os vértices amarelos são os próximos vizinhos não visitados que podem ser escolhidos. Os vértices em azul são pontos não visitados e que não são vizinhos do ponto atual.

Figura 04 - Ponto Inicial da **DFS**

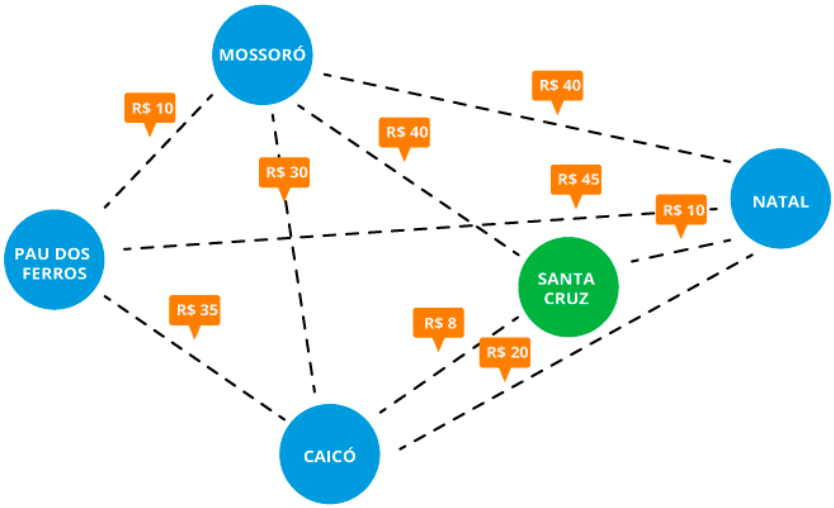


Figura 05 - Vizinhos que podem ser visitados a partir do vértice atual

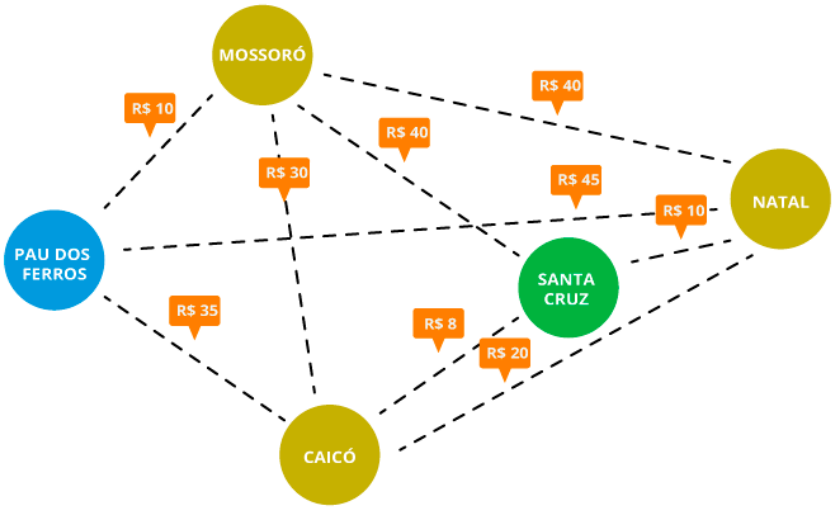


Figura 06 - Próximo vértice: Caicó

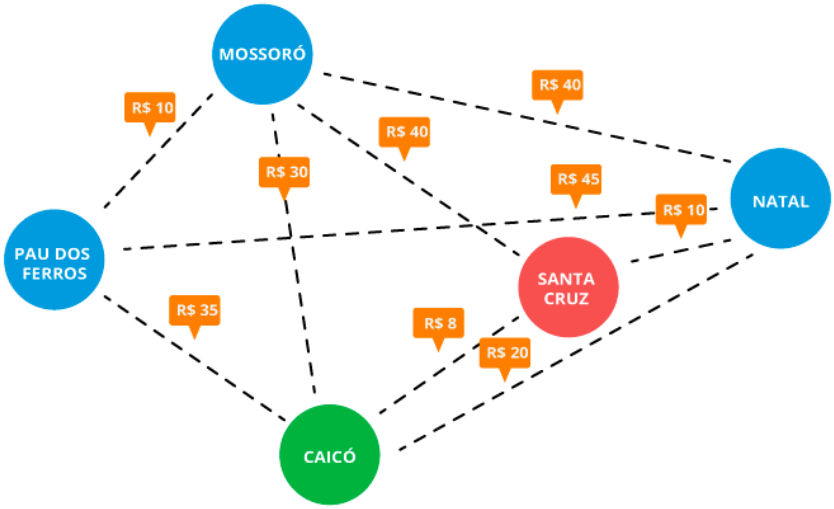


Figura 07 - Vizinhos que podem ser visitados a partir do vértice atual

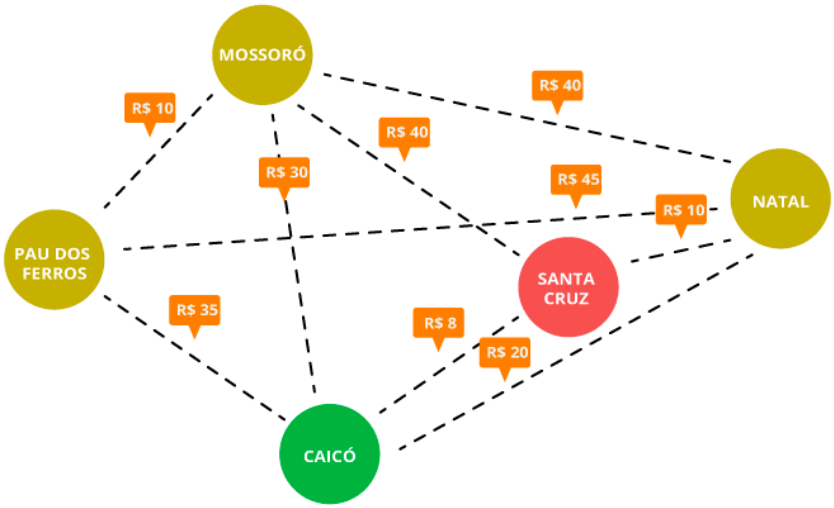


Figura 08 - Próximo vértice: Mossoró

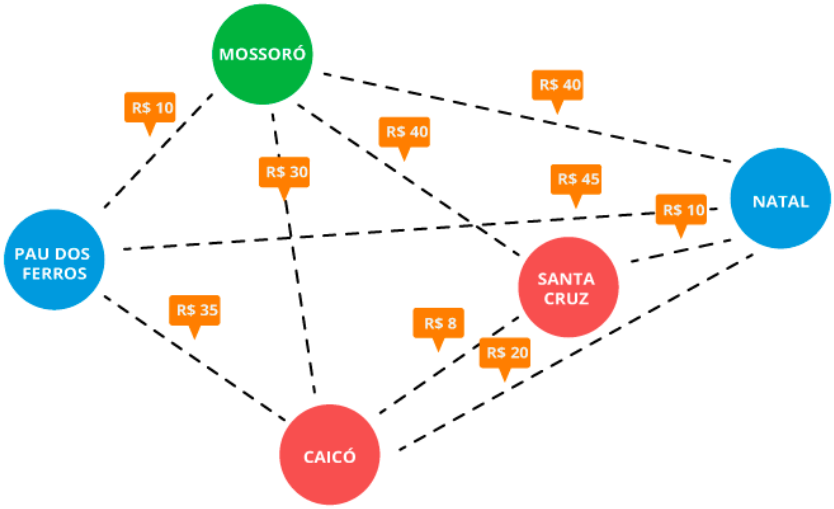


Figura 09 - Vizinhos que podem ser visitados a partir do vértice atual

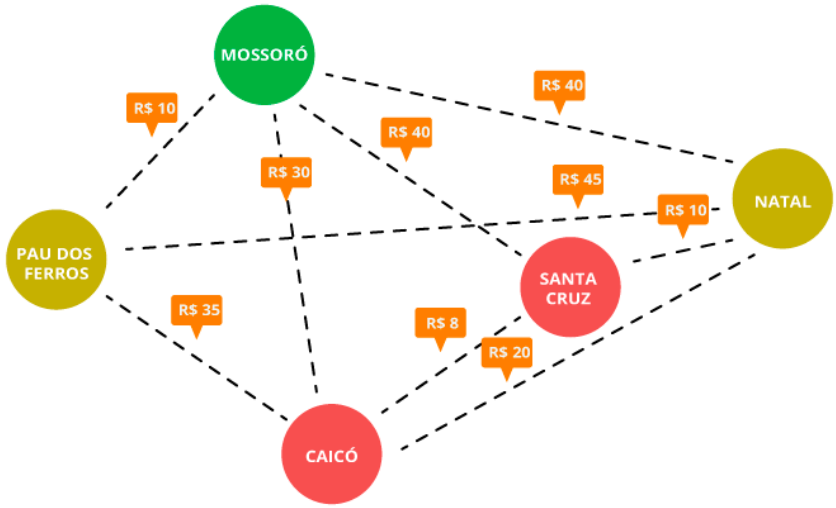
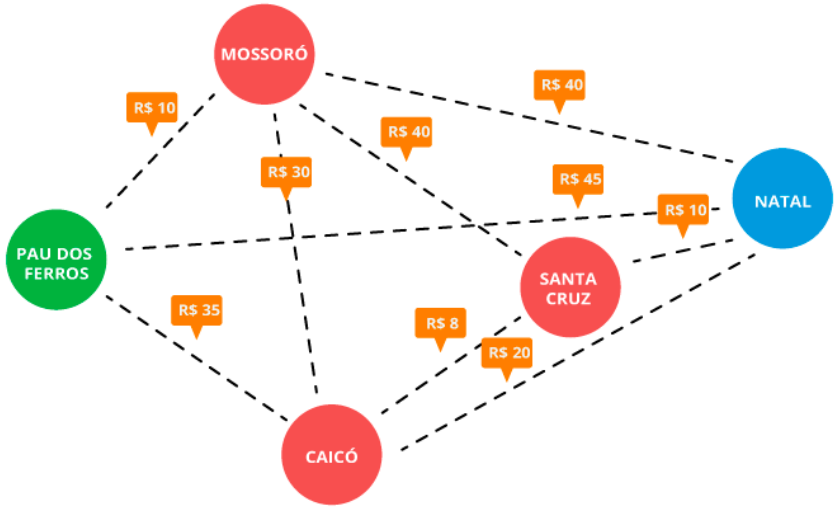


Figura 10 - Percurso chega ao destino



Uma observação a fazer é que a escolha do próximo vértice a ser explorada foi aleatória, você poderia ter feito um percurso distinto seguindo as regras e estaria correto da mesma forma! Nesse ponto, foi traçado um caminho em que você parte de Santa Cruz e passa por Caicó e Mossoró antes de chegar em Pau dos Ferros. Se você quiser interromper o algoritmo porque encontrou um caminho, sem problemas! Ou você pode continuar “rodando” para ver se encontra outros caminhos até o vértice final (vai que tem um melhor!). Nesse caso, o algoritmo vai

retornar ao vértice anterior (Mossoró) e colocar algum dos vizinhos ainda não explorados para tentar chegar a Pau dos Ferros novamente. Se no **grafo** não existirem mais caminhos, o algoritmo simplesmente termina.

Veja a sequência abaixo que mostra a execução do algoritmo no **grafo** voltando um nível (Mossoró).

Figura 11 - Vértice atual: Mossoró

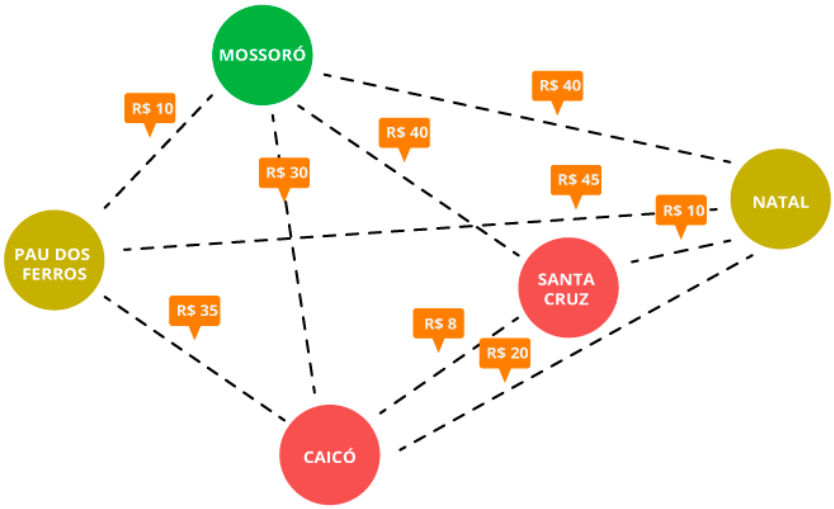


Figura 12 - Próximo vértice: Natal

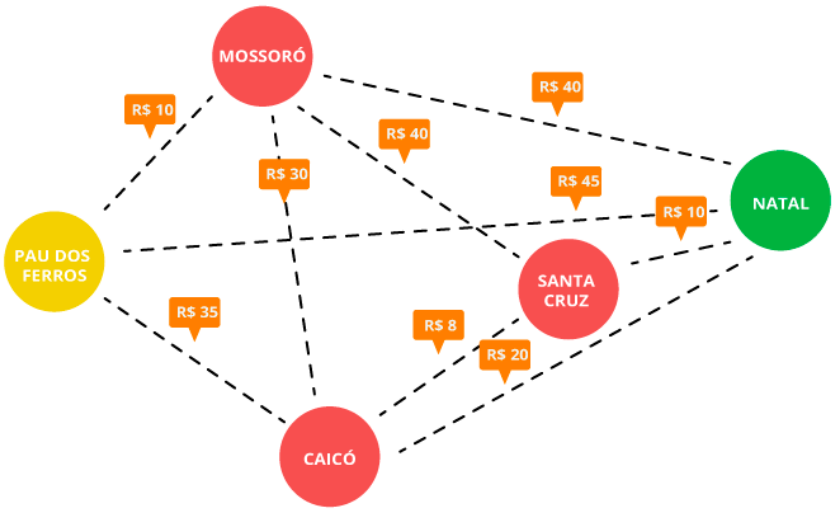
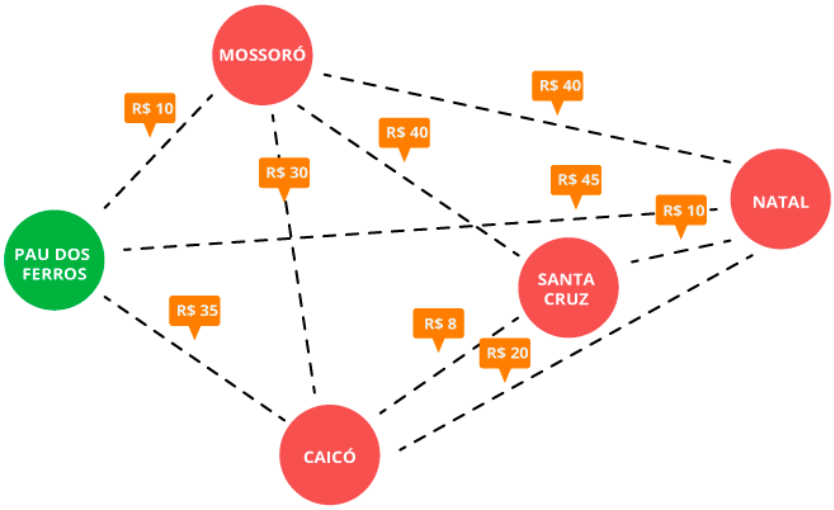


Figura 13 - Último vértice Pau dos Ferros



Nesse ponto, foi encontrado um caminho que passa por todos os vértices:

Santa Cruz -> Caicó -> Mossoró -> Natal -> Pau dos Ferros

E o que acontece se você quiser voltar dois níveis de profundidade, quando Caicó era a bola da vez? Você não poderá ir novamente para Mossoró, porque é um caminho que já foi explorado. Confira nas figuras abaixo.

Figura 14 - De volta para o futuro: Caicó

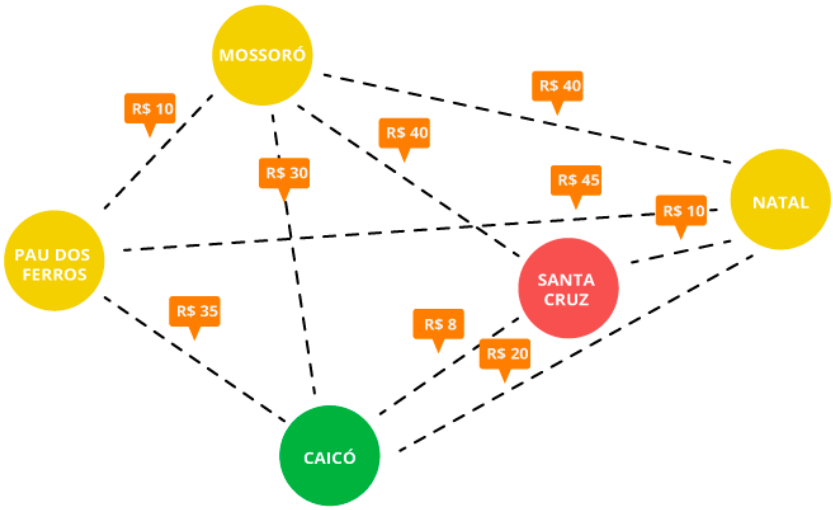
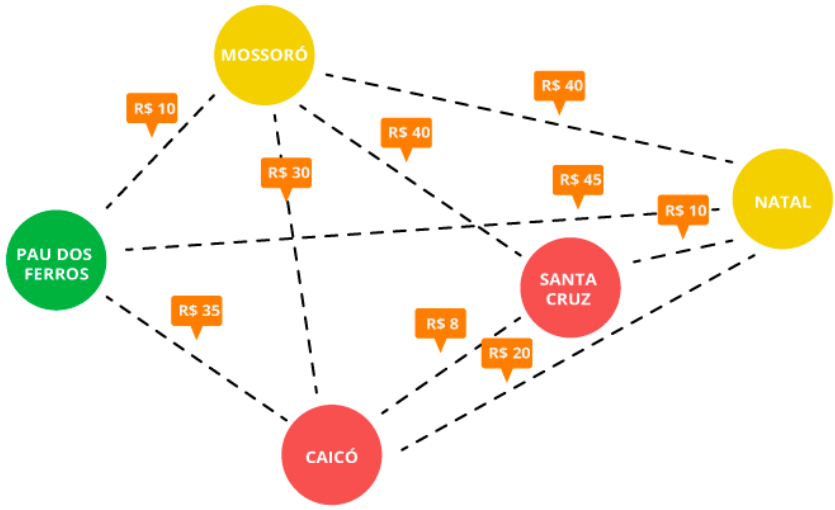


Figura 15 - Próximo vértice: Pau dos Ferros



Dessa forma, o caminho encontrado foi Santa Cruz, Caicó e Pau dos Ferros, uma rota mais direta! 😁 Esse algoritmo é uma recursão simples, mas é muito útil para várias situações na área de jogos. Imagine que você quer fazer um jogo que gera mapas procedurais: fazer um algoritmo que gera os mapas de qualquer jeito não é difícil (pode ser tudo aleatório), mas garantir que o mapa é válido, e que todas as casas são navegáveis, aí a conversa é mais séria... Uma estratégia válida é gerar um mapa e usar uma **DFS** para varrer e verificar se todas as casas podem ser alcançadas pelo jogador, adaptando o caminho quando ele encontrar falhas. O uso de algoritmos recursivos para explorar todas as possibilidades é uma técnica denominada ***Backtracking***.

Para não ficar apenas nos exemplos abstratos e visuais, veja um exemplo usando o **grafo** do RN, implementado em C# (Código 02).

Código 02 – Script C# para a **DFS**

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Algoritmos : MonoBehaviour {
    //matriz de adjacência para o grafo RN
    private int [,] grafo = new int[5,5] { {0, 40, 10, 20, 45},
                        {40, 0, 40, 30, 10},
                        {10, 40, 0, 8, -1},
                        {20, 30, 8, 0, 35},
                        {45, 10, -1, 35, 0}};

    //Nós que já foram visitados, inicia todos como não visitados
    private int[] visitados = new int[5]{0,0,0,0,0};

    //Nomes das cidades
   string[] cidades = new string[5]{"Natal", "Mossoró", "Santa Cruz", "Caicó", "Pau dos Ferros"};

    //Nó inicial - 0 representa Natal
    private int inicio = 0;

    //Nó final - 3 representa Caicó
    private int fim = 3;

    string caminho = "";

    void Start () {
     print("COMECANDO A BUSCA");
     Busca(inicio, fim);
    }

    // Update is called once per frame
    void Update () {}


    //Busca
    void Busca(int start, int finish){
     //Em profundidade
         DFS(start, finish);
    }

    //Algoritmo da busca em profundidade, para achar todos os caminhos
    void DFS(int atual, int alvo){
     string aux;
     //Monta string do caminho
     caminho += cidades[atual];
     aux = caminho;

     //Marca ela como visitada
     visitados[atual] = 1;

```

```
    //Se for a cidade destino
    if(atual == alvo){
        //Chegou bixiga!!
        print(caminho);
    }
    //Senão
    else{
        //Verifica quais são os vizinhos, ou nós que podem ser alcançados a partir do nó atual
        for(int i=0; i<5; i++){
            if(grafo[atual, i] > 0 && visitados[i] == 0){
                //Chama a função passando o vizinho como nó atual, adiciona uma seta para ficar bonita
                caminho += " -> ";

                DFS(i, alvo);

                //Na recursão, você desfaz as operações para preparar para executar o próximo passo co
                caminho = aux;
                visitados[i] = 0;
            }
        }
    }
   }
}
```

A **busca** em profundidade é um recurso muito utilizado, e que se vale de uma implementação recursiva para manter o código simples e elegante. Mas ela não é o único tipo de busca que pode explorar os vértices do **grafo**! Existe uma segunda abordagem chamada de **busca em largura**!

3 - Busca em Largura

Retome o exemplo da viagem de Santa Cruz para Pau dos Ferros, agora para entender o outro tipo de busca.

Na **busca em largura**, ou **BFS**, a ideia é que você explore todos os vizinhos do vértice em uma mesma profundidade antes de verificar os próximos vértices. No fim das contas você vai realizar os mesmos processamentos, só que em uma ordem diferente! Essa busca é mais sistemática e explora primeiro os vértices que estão mais próximos da origem, dirigindo-se gradativamente aos vértices mais distantes. É a ideia oposta da **busca em profundidade**: ao invés de tentar achar logo um caminho, ele vai verificando todos os caminhos do **grafo** até encontrar um que seja o destino desejado.

Figura 16 - Essa sim parece uma busca segura!

Para implementar essa ideia, parta para uma abordagem iterativa, utilizando uma Fila (lembra da Aula 03? A mesminha! 😁). Você vai pegar seu vértice de origem, Santa Cruz, e vai sair adicionando todos os vizinhos dele: Mossoró, Natal e Caicó. Todos esses vértices serão colocados em uma fila para serem processados. Após isso, o primeiro que foi colocado na fila torna-se o atual, e os seus vizinhos que ainda não estão na fila serão então inseridos no final. Perceba que esses novos vizinhos só serão processados quando todos os três vizinhos de Santa Cruz forem avaliados.

Dessa forma, você vai fazendo uma varredura do **grafo** pelos níveis de vértices que estão mais próximos do ponto de origem, e os nós mais distantes serão atingidos no final do algoritmo. Essa abordagem é boa para momentos em que você quer encontrar o primeiro caminho possível com o menor número de percursos no **grafo** (isso faz mais sentido quando o **grafo** não tem peso, em que o número de arestas do caminho seria equivalente ao seu custo!). O pseudocódigo desse algoritmo é ilustrado pelas linhas do Código 03 abaixo:

Código 01 – Pseudocódigo da **Busca em Largura**

```
1  Função BFS(int inicial, int destino){
2     Fila próximos
3
4     Insira o nó inicial na fila
5
6     Enquanto a fila não está vazia{
7        Processa o próximo da fila
8        Retira o próximo elemento da fila
9        Marca o elemento como visitado
10
11       Para cada vizinho do próximo{
12          Insere o vizinho na fila
13          Marca o vizinho como visitado
14       }
15    }
16 }
```

Hum... ainda está confuso, hein? Já sei, já sei, você quer as imagens... No caso da **busca em largura**, os vértices em amarelo vão representar os vértices inseridos na fila de avaliação pelo algoritmo, enquanto os outros mantêm o mesmo significado do que foi anteriormente visto no algoritmo **DFS**.

Figura 17 - Ponto inicial da **BFS**

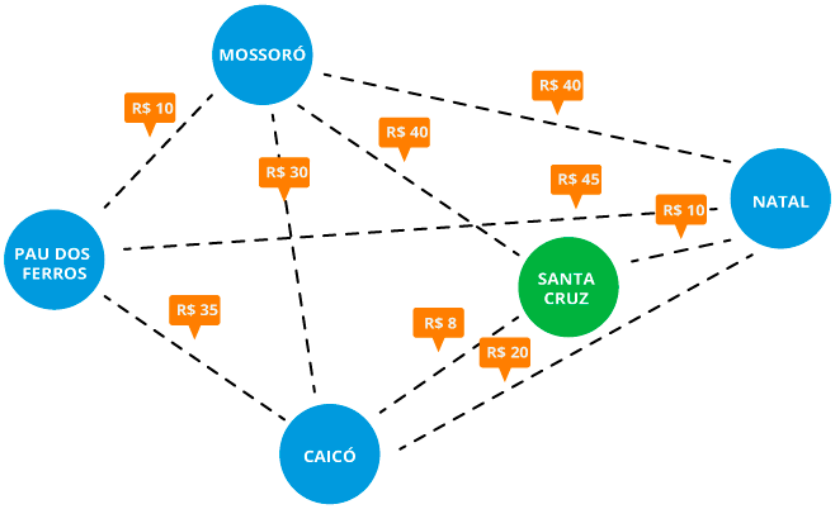


Figura 18 - Vizinhos de Santa Cruz: Caicó é inserido na fila

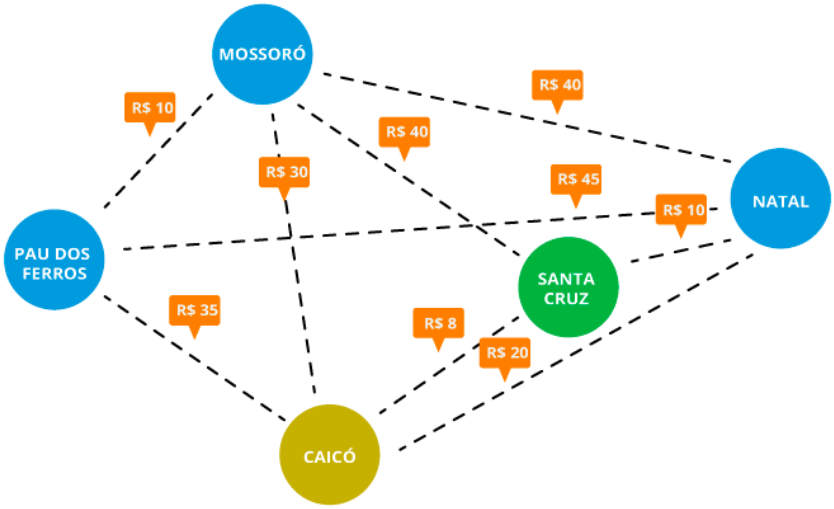


Figura 19 - Vizinhos de Santa Cruz: Mossoró é inserido na fila

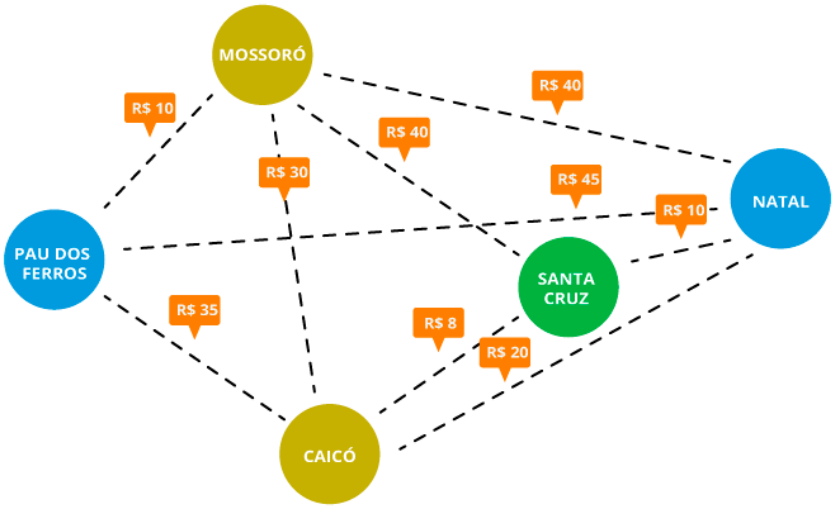


Figura 20 - Vizinhos de Santa Cruz: Natal é inserido na fila

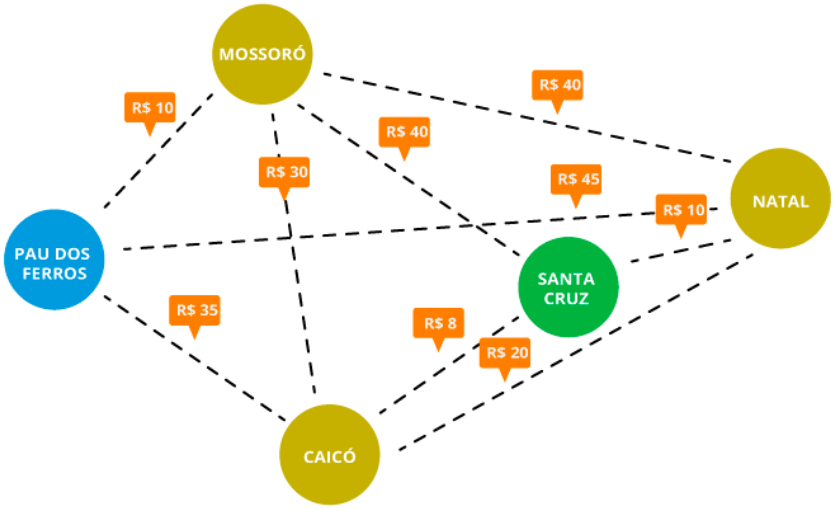


Figura 21 - Próximo vértice: Caicó

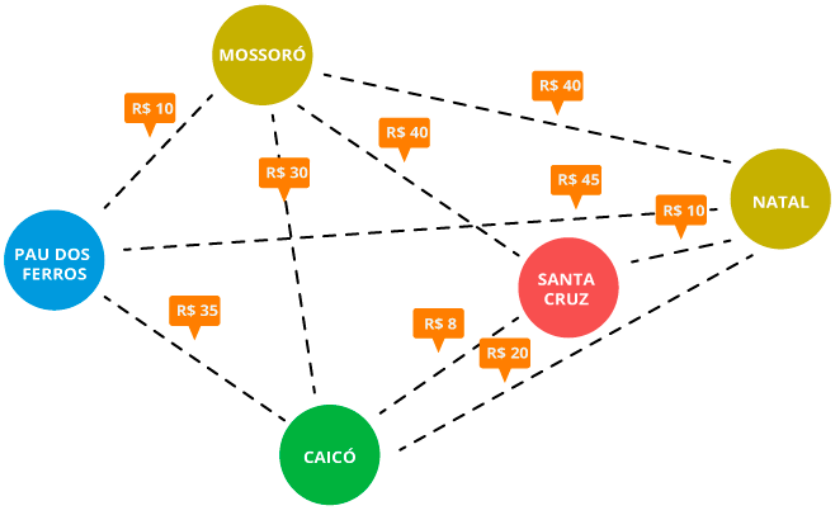


Figura 22 - Insere vizinhos não visitados de Caicó: Pau dos Ferros

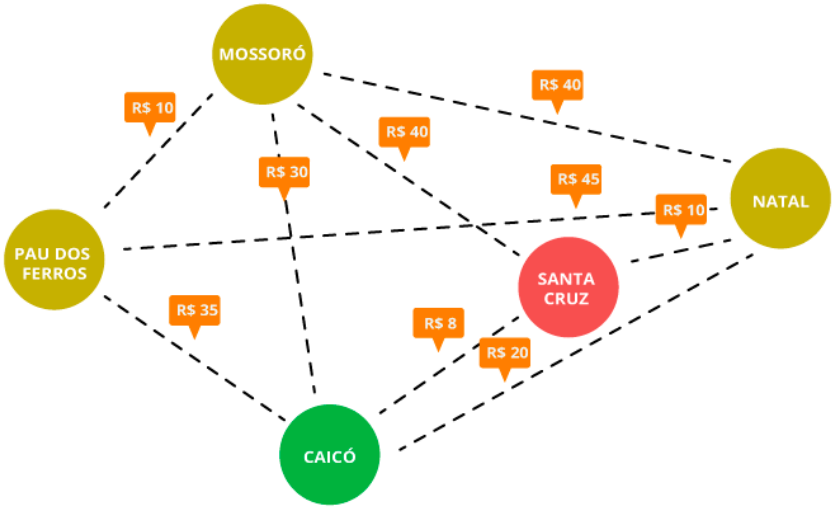


Figura 23 - Próximo vértice: Mossoró

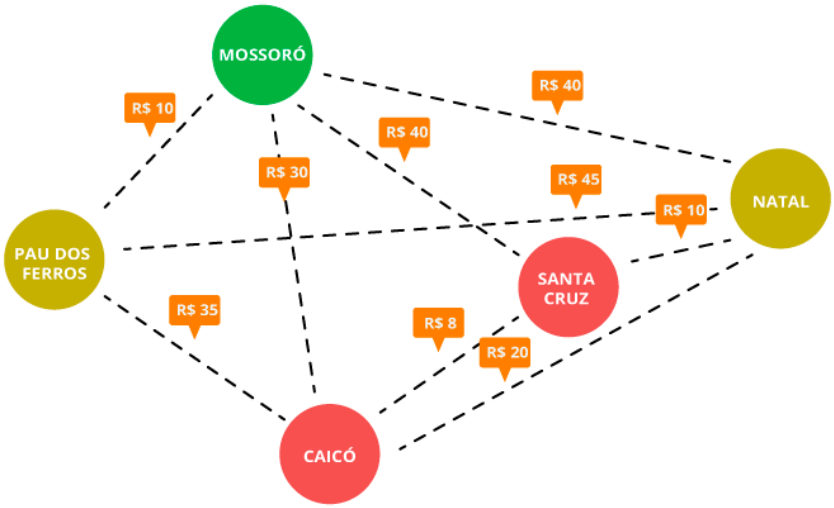


Figura 24 - Próximo Vértice: Natal

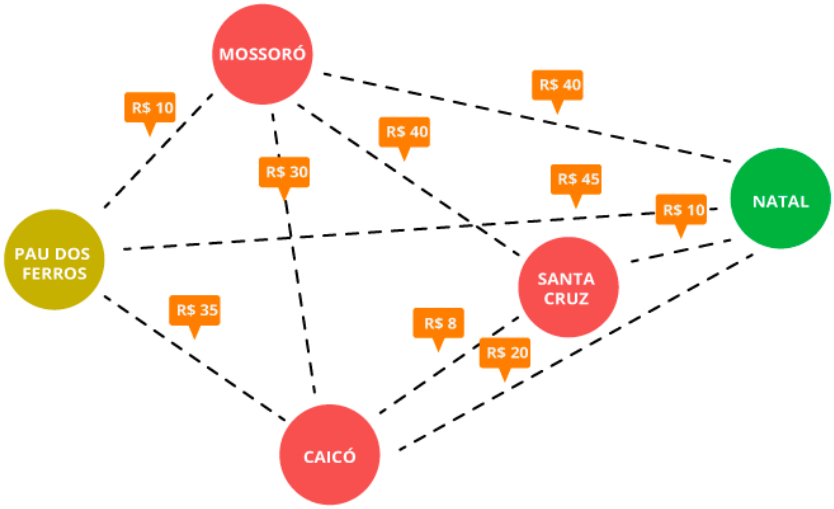
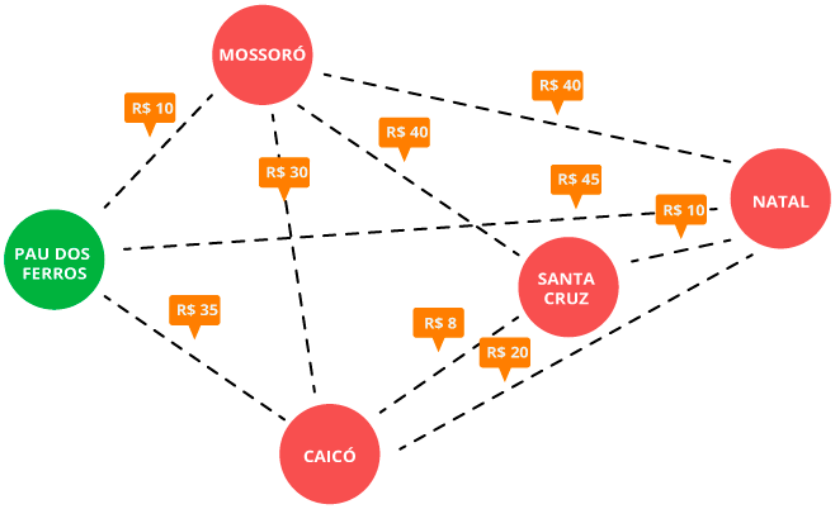


Figura 25 - Chegada ao destino desejado



Como você pôde observar na Figura 16, o caminho final dessa iteração foi chegar a Pau dos Ferros através de Caicó, vértice a partir do qual Pau dos Ferros foi inserido na fila. Da mesma forma que o algoritmo anterior, vou colocar um exemplo de implementação utilizando a linguagem C# (Código 04). Esse exemplo busca o primeiro caminho possível de um ponto inicial até um ponto final.

Código 04 – Script C# para a **BFS**

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Algoritmos : MonoBehaviour {
    //matriz de adjacência para o grafo RN
    private int [,] grafo = new int[5,5] { {0, 40, 10, 20, 45},
                        {40, 0, 40, 30, 10},
                        {10, 40, 0, 8, -1},
                        {20, 30, 8, 0, 35},
                        {45, 10, -1, 35, 0}};

    //Nós que já foram visitados, inicia todos como não visitados
    private int[] visitados = new int[5]{0,0,0,0,0};

    //Nomes das cidades
   string[] cidades = new string[5]{"Natal", "Mossoró", "Santa Cruz", "Caicó", "Pau dos Ferros"};

    //Nó inicial - 0 representa Natal
    private int inicio = 0;

    //Nó final - 3 representa Caicó
    private int fim = 3;

    string caminho = "";

    void Start () {
     print("COMECANDO A BUSCA");
     Busca(inicio, fim);
    }

    // Update is called once per frame
    void Update () {}


    //Busca
    void Busca(int start, int finish){
     //Em largura
     //BFS(start, finish)
    }

//Algoritmo BFS, montando para parar no primeiro caminho que encontrar
    void BFS(int atual, int alvo){
     //Fila de nós a serem processados
     Queue<int> fila = new Queue<int>();
     int prox, cont = 0;

     //Coloca o nó inicial na fila e marca como visitado
     fila.Enqueue(atual);
     visitados[atual] = 1;

```

```
52      //Enquanto a fila não secar
53      while(fila.Count > 0){
54          //Pega o próximo na fila
55          prox = fila.Peek();
56          visitados[prox] = 1;
57
58          if(cont != 0)
59              caminho += " -> ";
60
61          caminho += cidades[prox];
62
63          //Remove ele da fila
64          fila.Dequeue();
65
66          //Coloca os vizinhos na fila, e marca como visitado
67          for(int i=0; i<5; i++){
68              if(grafo[prox, i] > 0 && visitados[i] == 0){
69                  if(i == alvo)
70                      caminho += " -> " + cidades[i];
71                  else{
72                      fila.Enqueue(i);
73                  }
74
75                  visitados[i] = 1;
76              }
77          }
78
79          //Se chegou no fim
80          if(visitados[alvo] == 1)
81              break;
82
83          //Atualiza contador auxiliar da impressão
84          cont++;
85      }
86
87      //Imprime o primeiro caminho que encontrou
88      print(caminho);
89  }
```

E com isso você já fica com duas ferramentas poderosas para trabalhar. Apesar de suas aplicações na área de jogos, esses algoritmos não se limitam apenas a esse campo, podendo ser encontrados na resolução de diversos problemas na área de redes, circuitos, otimização de algoritmos, engenharia de software e vários outros campos da computação! Então é muito importante entender a ideia de funcionamento e brincar com a implementação desses algoritmos para aumentar o seu arsenal como programador. 😁

Perceba uma coisa: com esses algoritmos você consegue verificar se existe um caminho de A até B, e até achar todos os caminhos que existem entre os dois! Mas eles não são direcionados para achar o menor caminho: aquele que te leva com o menor custo possível de um ponto a outro - a principal problemática que foi discutida no início das aulas de “Descoberta de caminhos”!

Não se desespere! Mature um pouco o conteúdo dessa aula, que na próxima você terá o *gran finale*! Conhecerá dois algoritmos voltados para essas tarefas: o **algoritmo de Djikstra** e o **algoritmo A*!**

Não consegue conter a emoção, hein? ~~Clique aqui para ver o próximo episódio.~~

Até lá! 😉

Resumo

Nesta aula você conheceu duas abordagens distintas para realizar percursos em **grafos**: a **busca em profundidade** e a **busca em largura**.

Na **busca em profundidade**, você afunila uma sequência de vértices em um caminho até chegar no ponto desejado (se for possível!), fazendo várias idas e voltas a partir do ponto de origem do **grafo**. Essa abordagem é muito boa para implementar de forma recursiva, e adotar uma estratégia de ***backtracking*** para descobrir todos os caminhos possíveis entre dois vértices!

Na **busca em largura**, você observou um método mais sistemático de varredura do **grafo**, com o algoritmo partindo de um nó inicial e irradiando sua busca nos pontos mais próximos, expandindo gradativamente para os vértices mais distantes do **grafo**. Essa abordagem é muito boa para verificar se é possível chegar a um vértice específico do **grafo** a partir de diversos pontos distintos.

Ambos os algoritmos são eficientes em percorrer o **grafo**, mas nenhum deles leva em conta decisões de otimização necessárias para encontrar o caminho de menor custo entre dois vértices.

Na próxima aula você conhecerá dois algoritmos que são capazes de realizar essa tarefa.

Bons estudos!

Midiateca

Criando um gerador de labirintos (aplicação de **DFS**) - https://www.youtube.com/watch?v=z7wHZMB9YYs

Referências

CORMEN, Thomas H. et al. **Introduction to algorithms third edition**. Massachusetts: MIT Press,v. 23, p. 631-638, 2009.

FEOFILOFF, Paulo. **Busca em Profundidade** (DFS). Disponível em: <https://www.ime.usp.br/~pf/algoritmos_para_grafos/aulas/dfs.html>. Acesso em: 28 maio 2018.

FEOFILOFF, Paulo. **Busca em Largura** (BFS). Disponível em: <https://www.ime.usp.br/~pf/algoritmos_para_grafos/aulas/bfs.html>. Acesso em: 28 maio 2018.

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