UNIVERSIDADE FEDERAL DO RIO GRANDE

Centro de Ciências Computacionais – C3

Bacharelado Em Sistemas de Informação

**Linguagens de Programação**

135313 – Lucas Leal

134539 – Lucas Machado

149456 – Thiago Azor

132878 – Daniel Correa

# **Algoritmo A\***

Professor: Rafael Berri

Rio Grande/RS Novembro/2022.

# **Algoritmo A\***

Trabalho apresentado na disciplina de Linguagens de Programação do curso de Bacharelado em Sistemas de Informação da Universidade Federal do Rio Grande, Sob Orientação do Professor: Rafael Berri.

Rio Grande/RS Novembro/2022

1. **O Algoritmo A\*(A estrela):**

A finalidade de um algoritmo A\* é procurar resolver uma sequência de passos (pathfinding) em um menor caminho, levado de um dado nó inicial a um destino em uma estrutura de dados, como grafos e árvores. O algoritmo foi descrito pela primeira vez em 1968 por Peter Hart, Nils Nilsson, e Bertram Raphael. Na publicação deles, ele foi chamado de algoritmo A; usando este algoritmo com uma heurística apropriada atinge-se um comportamento ótimo, e passou a ser conhecido por A\*.

Ele é dado através de uma função ou iterador que para cada nó da estrutura fornece os nós que são seu vizinhos. Frequentemente os nós representam um conjunto de configurações possíveis. O algoritmo é iterativo e no início de cada iteração temos uma partição dos seus nós em dois conjuntos:

* vistos: nós que já foram encontrados pelo algoritmo.
* não vistos: nós que o algoritmo não tem a menor ideia que existem.

O conjunto de nós vistos é ainda particionado em:

* vistos não examinados: nós que foram descobertos, mas que ainda o algoritmo não terminou olhou com cuidado.
* vistos examinados: nós que foram descobertos e o algoritmo não tem mais nada a fazer com eles.

O conjunto de nós vistos não examinados é o coração do algoritmo. Esse conjunto é mantido em uma fila priorizada que é o segredo para o desempenho do algoritmo. Como tipicamente o número de nós é muito grande, infelizmente, não é possível manter o conjunto de nós examinados explicitamente. Desta forma, a divisão dos nós vistos em vistos não examinados e vistos examinados pode não ser muito séria, pode não ser uma partição. Podemos ter nós vistos que estão em vistos não examinados e vistos examinados o que atrapalha um pouco a metáfora e o consumo de tempo e espaço.

**O algoritmo é restrito a um tipo especifico de estrutura?**

Não. Como explicado acima, é preciso pegar todos os nodos vizinhos do atual, podendo ser implementado em vários tipos de dados. Em um grafo direcionado são todos os nodos "saindo" do atual; em um grafo não-direcionado são todos os nodos ligados ao atual.

**Como ele determina o custo entre as arestas ligadas nos vértices de um grafo?**

Isso é feito pela função de heurística. Essa função é muito importante, pois é ela que vai direcionar a busca para o caminho correto. Isso depende inteiramente de como o grafo é feito, não existe uma "regra universal" de como criar a função de heurística. Ela precisa receber por parâmetro dois nodos do grafo e retornar um número indicando quão longe esses dois nodos estão.

Por exemplo: Se o grafo representar um labirinto 2D quadrado, os índices da célula representada pelo nodo podem ser usados pra calcular a distância. A distância entre os nodos das células (1, 1) e (5, 4) pode ser calculada pela fórmula da distância euclidiana.

**Existe alguma formula matemática usada nesse algoritmo?**

Isso obviamente varia de grafo pra grafo, mas a maneira mais usada para expressar grafos simples de serem implementados é através da fórmula f(n) = g(n) + h(n). Onde g é a distância do nodo atual ao nodo examinado e h é a heurística implementada ao nó.

É de grande responsabilidade entender como o grafo é formado pra descobrir uma fórmula matemática pra função de heurística.

1. **Linguagens escolhidas:**

* **PHP:**

PHP (um acrônimo recursivo para "PHP: Hypertext Preprocessor", originalmente Personal Home Page) é uma linguagem interpretada livre, usada originalmente apenas para o desenvolvimento de aplicações presentes e atuantes no lado do servidor, capazes de gerar conteúdo dinâmico na World Wide Web. Figura entre as primeiras linguagens passíveis de inserção em documentos HTML, dispensando em muitos casos o uso de arquivos externos para eventuais processamentos de dados. O código é interpretado no lado do servidor pelo módulo PHP, que também gera a página web a ser visualizada no lado do cliente. A linguagem evoluiu, passou a oferecer funcionalidades em linha de comando, e além disso, ganhou características adicionais, que possibilitaram usos adicionais do PHP, não relacionados a web sites.

* **LUA**

A linguagem Lua tem um alto nível de portabilidade, é leve e apresenta alguns paradigmas da programação.

* É uma linguagem orientada a objetos;
* É uma linguagem que permite a programação funcional;
* É uma linguagem orientada a dados.

Lua tem uma tipagem de dados funcionando de forma dinâmica, ou seja, os valores jamais terão tratamento com um tipo errado, conforme ocorre com as linguagens não tipadas. Além disso, é uma linguagem interpretada e precisa de um compilador para executar trechos de códigos existentes.

Em Lua, também podemos ter um autogerenciamento da memória. Isso significa que ela pode ser gerenciada automaticamente, simplificando processos como a criação de interfaces e prevenção de erros que possam ocorrer na alocação de memória, erros complexos e de difícil resolução.

* **C#**
* O C# (pronuncia-se "C Sharp") é uma linguagem de programação moderna, orientada a objeto e fortemente tipada. O C# permite que os desenvolvedores criem muitos tipos de aplicativos seguros e robustos que são executados no .NET. O C# tem suas raízes na família de linguagens C e os programadores em C, C++, Java e JavaScript a reconhecerão imediatamente. Este tour dá uma visão geral dos principais componentes da linguagem em C# 8 e anterior. Se quiser explorar a linguagem por meio de exemplos interativos, experimente os tutoriais de Introdução à linguagem C#.
* C# é uma linguagem de programação orientada a objetos e orientada a componentes. C# fornece construções de linguagem para dar suporte diretamente a esses conceitos, tornando
* C# uma linguagem natural para criação e uso de componentes de software. Desde sua origem, o C# adicionou recursos para dar suporte a novas cargas de trabalho e práticas emergentes de design de software. Em sua essência, o C# é uma linguagem orientada a objeto. Você define os tipos e o comportamento deles.

1. **Implementação do algoritmo:**

* **PHP**
* <?php
* class Graph {
* private array $adjacencyList;
* public function \_\_construct(array $adjacencyList) {
* *$this*->adjacencyList = $adjacencyList;
* }
* public function getNeighbors(string $node) {
* return *$this*->adjacencyList[$node];
* }
* public function nodeWeight(string $node) {
* $H = [
* 'A' => 1,
* 'B' => 1,
* 'C' => 1,
* 'D' => 1
* ];
* return $H[$node];
* }
* public function aStarAlgorithm(string $startNode, string $stopNode) {
* $openList = [$startNode];
* $closedList = [];
* $g = [];
* $g[$startNode] = 0;
* $parents = [];
* $parents[$startNode] = $startNode;
* while (count($openList) > 0) {
* $currentNode = null;
* // verifica qual é o menor nó vizinho do nó atual.
* foreach ($openList as $v) {
* if ($currentNode == null || $g[$v] + *$this*->nodeWeight($v) <
* $g[$currentNode] + *$this*->nodeWeight($currentNode)) {
* $currentNode = $v;
* }
* }
* if ($currentNode == null) {
* throw new *Exception*('Caminho não existe!!');
* }
* // quando acha o nó refaz o caminho.
* if ($currentNode == $stopNode) {
* $reconstPath = [];
* while ($parents[$currentNode] != $currentNode) {
* $reconstPath[] = $currentNode;
* $currentNode = $parents[$currentNode];
* }
* $reconstPath[] = $startNode;
* $reconstPath = array\_reverse($reconstPath);
* echo sprintf('caminho encontrado: %s', implode(' -> ', $reconstPath));
* return true;
* }
* // adiciona o custo do caminho do nó vizinho em g e relaciona ao nó atual.
* foreach (*$this*->getNeighbors($currentNode) as $item) {
* $m = $item[0];
* $weight = $item[1];
* if (!in\_array($m, $openList) && !in\_array($m, $closedList)) {
* $openList[] = $m;
* $parents[$m] = $currentNode;
* $g[$m] = $g[$currentNode] + $weight;
* }
* // verifica se é melhor voltar ou passar para o proximo nó atraves do peso.
* else {
* if ($g[$m] > $g[$currentNode] + $weight) {
* $g[$m] = $g[$currentNode] + $weight;
* $parents[$m] = $currentNode;
* if (in\_array($m, $closedList)) {
* $closedList = array\_filter($closedList, fn($node) => $node != $m);
* $openList[] = $m;
* }
* }
* }
* }
* $openList = array\_filter($openList, fn($node) => $node != $currentNode);
* $closedList[] = $currentNode;
* }
* echo 'Caminho não existe';
* return null;
* }
* }
* $adjacencyList = [
* 'A' => [['B', 1], ['C', 3], ['D', 7]],
* 'B' => [['D', 5]],
* 'C' => [['D', 12]]
* ];
* $Graph1 = new *Graph*($adjacencyList);
* $Graph1->aStarAlgorithm('A', 'D');

Imagem 1 – implementação em PHP

* **LUA**
* Graph = {adjacency\_list = {}}
* function Graph.get\_neighbors(*self*, *node*)
* return *self*.adjacency\_list[node]
* end
* function Graph.node\_weight(*self*, *node*)
* H = {}
* H['A'] = 1
* H['B'] = 1
* H['C'] = 1
* H['D'] = 1
* return H[node]
* end
* function Graph.a\_star\_algorithm(*self*, *start\_node*, *stop\_node*)
* open\_list = {start\_node}
* closed\_list = {}
* g = {}
* g[start\_node] = 0
* --verifica
* parents = {}
* parents[start\_node] = start\_node
* while #open\_list > 0 do
* current\_node = nil
* for \_, v in ipairs (open\_list) do
* if current\_node == nil or g[v] + *self*.node\_weight(*self*, v) <
* g[current\_node] + *self*.node\_weight(*self*, current\_node) then
* current\_node = v
* end
* end
* if current\_node == nil then
* print('Caminho não existe!')
* return nil
* end
* -- quando acha o nó refaz o caminho.
* if current\_node == stop\_node then
* reconst\_path = {}
* while parents[current\_node] ~= current\_node do
* table.insert(reconst\_path, current\_node)
* current\_node = parents[current\_node]
* end
* table.insert(reconst\_path, start\_node)
* array\_reverse(reconst\_path)
* reconst\_path\_str = ''
* for i, node in ipairs (reconst\_path) do
* reconst\_path\_str = reconst\_path\_str .. node
* if i < #reconst\_path then
* reconst\_path\_str = reconst\_path\_str .. ' -> '
* end
* end
* print('caminho encontrado: ' .. reconst\_path\_str)
* return true
* end
* -- adiciona o custo do caminho do nó vizinho em g e relaciona ao nó atual.
* for \_, item in ipairs (*self*.get\_neighbors(*self*, current\_node)) do
* m = item[1]
* weight = item[2]
* if not\_in(open\_list, m) and not\_in(closed\_list, m) then
* table.insert(open\_list, m)
* parents[m] = current\_node
* g[m] = g[current\_node] + weight
* -- verifica se é melhor voltar ou passar para o proximo nó atraves do peso.
* else
* if g[m] > g[current\_node] + weight then
* g[m] = g[current\_node] + weight
* parents[m] = current\_node
* if has\_value(closed\_list, m) then
* remove\_node(closed\_list, m)
* table.insert(open\_list, m)
* end
* end
* end
* end
* remove\_node(open\_list, current\_node)
* table.insert(closed\_list, current\_node)
* end
* print('Caminho não existe')
* return nil
* end
* function array\_reverse(*x*)
* local n, m = #x, #x/2
* for i=1, m do
* x[i], x[n-i+1] = x[n-i+1], x[i]
* end
* return x
* end
* function remove\_node (*set*, *theNode*)
* for i, node in ipairs (set) do
* if node == theNode then
* set [ i ] = set [ #set ]
* set [ #set ] = nil
* break
* end
* end
* end
* function has\_value (*tab*, *val*)
* for index, value in ipairs(tab) do
* if value == val then
* return true
* end
* end
* return false
* end
* function not\_in ( *set*, *theNode* )
* for \_, node in ipairs ( set ) do
* if node == theNode then return false end
* end
* return true
* end
* adjacency\_list = {}
* adjacency\_list['A'] = {}
* adjacency\_list['B'] = {}
* adjacency\_list['C'] = {}
* table.insert(adjacency\_list['A'], {'B', 1})
* table.insert(adjacency\_list['A'], {'C', 3})
* table.insert(adjacency\_list['A'], {'D', 7})
* table.insert(adjacency\_list['B'], {'D', 5})
* table.insert(adjacency\_list['C'], {'D', 12})
* Graph.adjacency\_list = adjacency\_list
* Graph.a\_star\_algorithm(Graph, 'A', 'D')

Imagem 2 - Implementação em LUA

* **C#**
* using *System*;
* using *System*.*Collections*;
* using *System*.*Collections*.*Generic*;
* using *System*.*Linq*;
* using *System*.*Text*;
* using *System*.*Numerics*;
* namespace *AStarSharp*
* {
* public class Node
* {
* public static *int* NODE\_SIZE = 32;
* public *Node* Parent;
* public *Vector2* Position;
* public *Vector2* Center
* {
* get
* {
* return new *Vector2*(Position.X + NODE\_SIZE / 2, Position.Y + NODE\_SIZE / 2);
* }
* }
* public *float* DistanceToTarget;
* public *float* Cost;
* public *float* Weight;
* public *float* F
* {
* get
* {
* if (DistanceToTarget != -1 && Cost != -1)
* return DistanceToTarget + Cost;
* else
* return -1;
* }
* }
* public *bool* Walkable;
* public Node(*Vector2* *pos*, *bool* *walkable*, *float* *weight* = 1)
* {
* Parent = null;
* Position = pos;
* DistanceToTarget = -1;
* Cost = 1;
* Weight = weight;
* Walkable = walkable;
* }
* }
* public class Astar
* {
* *List*<*List*<*Node*>> Grid;
* *int* GridRows
* {
* get
* {
* return Grid[0].Count;
* }
* }
* *int* GridCols
* {
* get
* {
* return Grid.Count;
* }
* }
* public Astar(*List*<*List*<*Node*>> *grid*)
* {
* Grid = grid;
* }
* public *Stack*<*Node*> FindPath(*Vector2* *Start*, *Vector2* *End*)
* {
* *Node* start = new *Node*(new *Vector2*((*int*)(Start.X/Node.NODE\_SIZE), (*int*) (Start.Y/Node.NODE\_SIZE)), true);
* *Node* end = new *Node*(new *Vector2*((*int*)(End.X / Node.NODE\_SIZE), (*int*)(End.Y / Node.NODE\_SIZE)), true);
* *Stack*<*Node*> Path = new *Stack*<*Node*>();
* *PriorityQueue*<*Node*, *float*> OpenList = new *PriorityQueue*<*Node*,*float*>();
* *List*<*Node*> ClosedList = new *List*<*Node*>();
* *List*<*Node*> adjacencies;
* *Node* current = start;
* // adiciona o nó inicial na openlist.
* OpenList.Enqueue(start, start.F);
* while(OpenList.Count != 0 && !ClosedList.Exists(*x* => x.Position == end.Position))
* {
* current = OpenList.Dequeue();
* ClosedList.Add(current);
* adjacencies = GetAdjacentNodes(current);
* foreach(*Node* n in adjacencies)
* {
* if (!ClosedList.Contains(n) && n.Walkable)
* {
* *bool* isFound = false;
* foreach (var oLNode in OpenList.UnorderedItems)
* {
* if (oLNode.Element == n)
* {
* isFound = true;
* }
* }
* if (!isFound)
* {
* n.Parent = current;
* n.DistanceToTarget = Math.Abs(n.Position.X - end.Position.X) + Math.Abs(n.Position.Y - end.Position.Y);
* n.Cost = n.Weight + n.Parent.Cost;
* OpenList.Enqueue(n, n.F);
* }
* }
* }
* }
* // constrói o caminho do algoritmo
* if(!ClosedList.Exists(*x* => x.Position == end.Position))
* {
* return null;
* }
* *Node* temp = ClosedList[ClosedList.IndexOf(current)];
* if (temp == null) return null;
* do
* {
* Path.Push(temp);
* temp = temp.Parent;
* } while (temp != start && temp != null) ;
* return Path;
* }
* private *List*<*Node*> GetAdjacentNodes(*Node* *n*)
* {
* *List*<*Node*> temp = new *List*<*Node*>();
* *int* row = (*int*)n.Position.Y;
* *int* col = (*int*)n.Position.X;
* if(row + 1 < GridRows)
* {
* temp.Add(Grid[col][row + 1]);
* }
* if(row - 1 >= 0)
* {
* temp.Add(Grid[col][row - 1]);
* }
* if(col - 1 >= 0)
* {
* temp.Add(Grid[col - 1][row]);
* }
* if(col + 1 < GridCols)
* {
* temp.Add(Grid[col + 1][row]);
* }
* return temp;
* }
* }
* }

Imagem 3 – Implementação em C#

1. **Exemplo de entrada e saída:**

Na linguagem PHP, instanciamos um objeto grafo através das seguintes linhas.

$adjacencyList = [

    'A' => [['B', 1], ['C', 3], ['D', 7]],

    'B' => [['D', 5]],

    'C' => [['D', 12]]

];

$Graph1 = new *Graph*($adjacencyList);

$Graph1->aStarAlgorithm('A', 'D');

Imagem 4 – criação do grafo em PHP

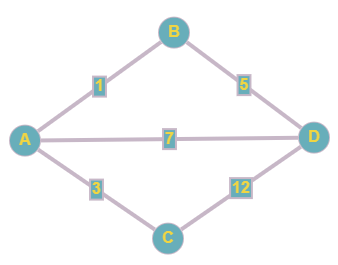


Imagem 5 – Representação física do grafo

O metódo “a\_star\_algorithm” recebe dois parâmetros, sendo eles, nó inicial (‘A’) do qual estamos partindo e o nó final (‘D) que é o objetivo de chegada. Após a chamada da função recebemos os seguintes dados de saída:

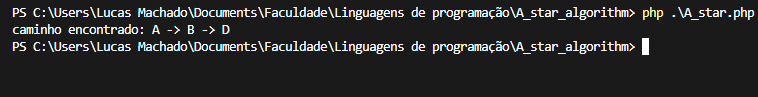


Imagem 6 – retorno da função

1. **Pesquisa sobre facilidade de leitura:**

Para compreender a facilidade da leitura dos algoritmos da pesquisa, utilizamos a ferramenta google forms para a coleta dos dados gerados, onde foi enviado para colegas que estão estudando e trabalhando na área de desenvolvimento. Um total de 16 pessoas responderam o formulário.

Gráficos obtidos durante a pesquisa:

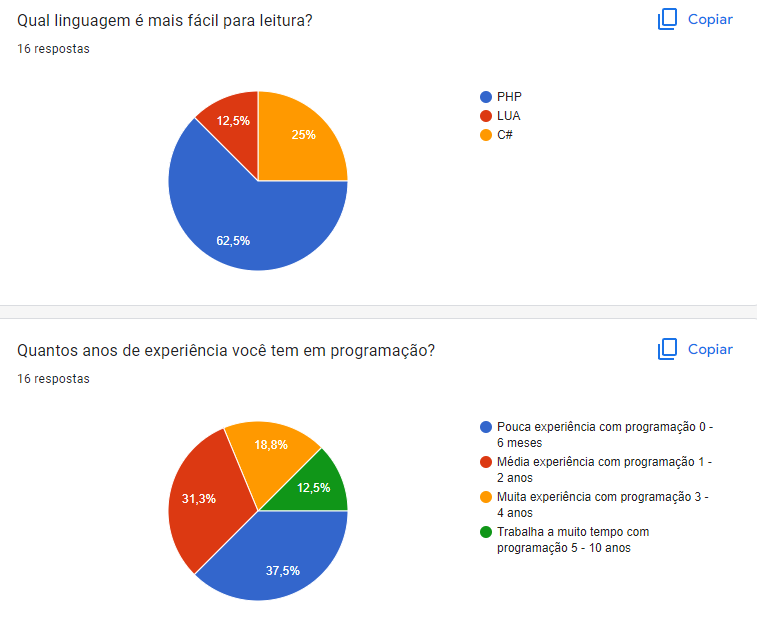


Imagem 7 – gráficos de facilidade e experiência

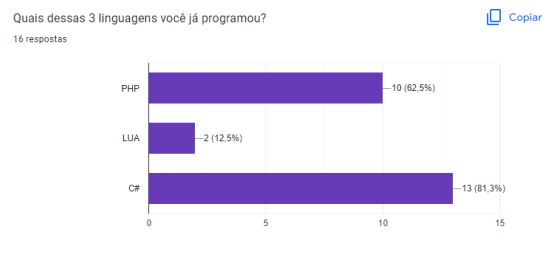


Imagem 8 – gráfico linguagens que o usuário já programou

1. **Conclusão:**

PHP foi a linguagem mais votada em questão de facilidade de leitura do código, seguido de C# e lua. Mesmo que C# foi a linguagem que os usuários mais utilizaram no desenvolvimento de códigos, percebesse que a syntax da linguagem influência muito na leitura. Lua possui uma syntax relativamente mais fácil em comparação com as outras linguagens porém usuários tem mais dificuldade de analisar o código, possui também uma popularidade menor de acordo com o site Stack overflow.