Métodos de Busca

F. P. Coelho[1] and T. F. Carvalho[2]
[1,3]Instituto de Ciênicas Tecnológicas, Universidade Federal de Itajubá, Itabira, Minas Gerais, Brasil
felipewallacepc@unifei.edu.br, thiago.carvalho@unifei.edu.br

I. Introdução

Os métodos de busca são amplamente utilizados na inteligência artificial para encontrar soluções para problemas complexos. Eles são uma técnica sistemática que podem examinar todas ou as possíveis soluções em um espaço de busca dado um determinado problema. A busca heurística, por sua vez, utiliza informações adicionais para guiar a busca e reduzir o espaço de busca, enquanto a busca não informada explora todo o espaço de busca.

Neste artigo, serão discutidos os principais métodos de busca e busca heurística, tais como a busca em profundidade, a busca em largura e o algoritmo A*. Serão abordadas suas complexidades de tempo e espaço, bem como suas limitações e possíveis aplicações.

II. Pesquisa sobre métodos de busca

A busca é uma técnica sistemática amplamente utilizada na inteligência artificial para encontrar soluções para problemas complexos. Ela examina possíveis soluções em um espaço de busca específico. Existem dois tipos principais de busca: a não informada, que não se orienta por informações adicionaus, e a informada, que emprega informações adicionais para reduzir o espaço de busca. Nesta seção, iremos explorar alguns dos tipos mais comuns de busca não informada e informada e como eles são aplicados na solução de problemas.

A. Busca não informada

A busca não informada, também conhecidas como busca em espaço de estados, consiste em algoritmos que não utilizam conhecimento específico do domínio para guiar o processo de busca. Esses algoritmos exploram sistematicamente o espaço de busca, sem qualquer informação adicional sobre o domínio do problema.

Existem vários tipos de estratégias de busca não informadas, incluindo busca em profundidade, em largura e busca em aprofundamento iterativo. Estas e outras buscas serão melhor explicadas a seguir.

Busca em Largura (BFS): É uma estratégia que explora o grafo em largura, ou seja, visita todos os nós de um mesmo nível antes de explorar os níveis seguintes. Essa técnica é garantida para encontrar a solução mais curta em um grafo não ponderado, mas pode não ser eficiente em grafos grandes ou com muitas ramificações. A complexidade de tempo da BFS é $O(b^d)$, onde b é o fator de ramificação e d é a profundidade da solução.

978-1-7281-7539-3/20/$31.00 ©2020 IEEE

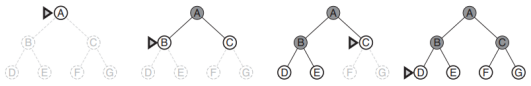

Figura 1. Busca em largura em uma árvore binária simples. Em cada etapa, o nó a ser expandido a seguir é indicado por um marcador. Adaptado de RUSSELL e NORVIG (2010).

Busca em Profundidade (DFS): É uma estratégia que explora o grafo em profundidade, visitando todos os nós de um ramo antes de retroceder para o nó anterior. Essa técnica pode ser implementada de forma recursiva, mas pode ficar presa em caminhos longos ou em ciclos infinitos. A complexidade de tempo da DFS é $O(b^m)$, onde b é o fator de ramificação e m é o comprimento máximo de um caminho.

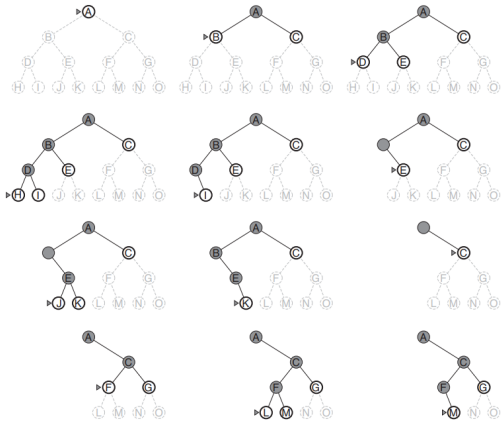

Figura 2. Busca em profundidade em uma árvore binária. A região não explorada é mostrada em cinza claro. Nós explorados sem descendentes na fronteira são removidos da memória. Nós na profundidade 3 não têm sucessores e M é o único nó objetivo. Adaptado de RUSSELL e NORVIG (2010).

Busca em Profundidade Limitada: É uma estratégia que limita a profundidade máxima da árvore de busca. Essa técnica é útil quando a profundidade máxima da solução é conhecida, mas pode falhar se o limite de profundidade for muito pequeno. A complexidade de tempo da busca com limite de profundidade é $O(b^l)$, onde l é o limite de profundidade.

Busca com Aprofundamento Iterativo (IDS): A busca com aprofundamento iterativo é uma estratégia que combina a busca em profundidade com a busca com limite de profundidade. Essa técnica itera a busca em profundidade com um limite de profundidade crescente em cada iteração. Essa técnica é garantida para encontrar a solução mais curta em grafos não ponderados e pode ser mais eficiente do que a busca em

largura em grafos grandes ou com muitas ramificações. A complexidade de tempo da IDS é $O(b^d)$, onde b é o fator de ramificação e d é a profundidade da solução.

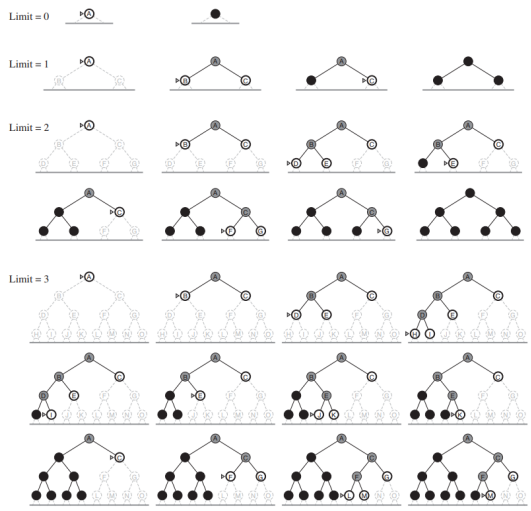

Figura 3. Quatro iterações da busca em profundidade iterativa em uma árvore binária. Adaptado de RUSSELL e NORVIG (2010).

A Tabela I compara as estratégias de busca em termos dos quatro critérios de avaliação, para o pior caso (*big-O*).

Tabela I
COMPLEXIDADE DOS ALGORITMOS DE BUSCA

	Tempo	**Espaço**	**Otima?**	**Completa?**
Largura	$O(b^d)$	$O(bd)$	Sim	Sim
Profundidade	$O(b^m)$	$O(bm)$	Não	Sim/Não*
Profundidade limitada	$O(b^l)$	$O(bl)$	Não	Sim se $l \geq d$
Profundidade iterativa	$O(b^d)$	$O(bd)$	Sim	Sim

b = número de caminhos alternativos/fator de bifurcação/ramificação; d = profundidade da solução; m = profundidade máxima da árvore de busca; l = limite de profundidade. $*$ = Sim (espaço de busca finito) / Não (espaço de busca infinito)
Fonte: Adaptado de RUSSELL, Stuart; NORVIG, Peter. Artificial Intelligence: A Modern Approach. 3. ed. Upper Saddle River, NJ: Pearson, 2010.

B. Busca Informada

Busca informada é uma técnica que utiliza informações adicionais sobre o problema para direcionar a busca em direção a soluções mais promissoras. Essas informações são incorporadas em uma função heurística, que é usada para avaliar a qualidade de cada solução candidata.

A busca informada é geralmente mais eficiente do que a busca cega, que explora o espaço de busca sem usar nenhuma informação adicional. Algumas técnicas populares de busca informada incluem a busca gulosa, A* e a busca de custo uniforme.

Busca Gulosa: O algoritmo guloso é uma técnica de otimização que se baseia em fazer escolhas locais ótimas em cada etapa da busca por uma solução global ótima. Essa estratégia pode levar a soluções de alta qualidade, mas não garante que a solução encontrada seja a melhor possível.

O algoritmo guloso pode ser aplicado em uma ampla variedade de problemas de otimização, desde problemas de programação linear até problemas de otimização de rotas. A escolha da função de avaliação e da estratégia de escolha de subproblemas é crucial para o desempenho do algoritmo.

Apesar de sua simplicidade, o algoritmo guloso é considerado uma técnica poderosa e eficiente para muitos problemas de otimização, mas pode falhar em certos casos. Para contornar essas limitações, podemos usar outras técnicas de otimização, como a busca A*.

Busca A*: É um algoritmo que utiliza o custo real do caminho percorrido até o momento e uma função heurística para estimar o custo restante até o objetivo. Ele expande os nós com o menor custo total (soma do custo percorrido e do custo estimado até o objetivo).

Podemos dizer que o algoritmo a-estrela é mais completo que a busca gulosa, pois ele considera tanto a estimativa do custo até o objetivo quanto o custo real do caminho percorrido até o momento, o que geralmente resulta em melhores soluções (como será visto na próxima seção).

Busca de Custo Uniforme: É uma variação do algoritmo de busca em largura. Nele iremos considerar o custo do caminho para chegar a um nó (parecido com o busca gulosa) em vez de usar apenas sua profundidade. O algoritmo é implementado através de uma fila de prioridades na qual cada posição da fila é ordenada pelo custo do caminho para chegar em cada nó. O algoritmo então expande o nó com o menor custo de caminho e atualiza as informações dos nós na fila de prioridade, caso uma rota mais barata seja encontrada.

Podemos observar que a busca de custo uniforme no geral é ótima. Pois sempre que a busca seleciona um nó n para ser expandido, o caminho ótimo até esse nó foi encontrado, caso não fosse verdade haveria outro nó que teria uma conexão com n' que conteria o caminho ótimo até n.

Esse tipo não se importa com o numero de "passos"que o melhor caminho terá, ele irá avaliar somente o custo total do caminho, mas ele irá ficar condicionado em um loop infinito caso um caminho possua sequências infinitas de caminhos que tenha um custo zero.

III. IMPLEMENTAÇÃO DE BUSCAS HEURÍSTICAS

Esta seção descreve uma implementação de duas buscas heurísticas, incluindo um resumo sobre o problema a ser resolvindo, uma breve descrição da Busca Gulosa e A* e quais estruturas de dados foram usadas na implementação.

A. Problema

Russell and Norvig [2010] propuseram um problema de otimização de rota que pode ser solucionado utilizando buscas heurísticas. O problema consiste em encontrar a rota mais curta entre duas cidades na Romênia, representadas por um grafo onde cada vértice é uma cidade e cada aresta tem a distância em milhas entre as cidades. Para solucionar o problema, é utilizada uma heurística que estima a distância em linha reta

entre a cidade atual e a cidade de destino. Para isso, foram fornecidas informações sobre as distâncias em linha reta entre cada cidade e a cidade de Bucharest. O algoritmo de busca gulosa utiliza apenas a heurística de distância estimada em linha reta, enquanto o algoritmo A* utiliza a heurística de distância estimada em linha reta somada à distância local.

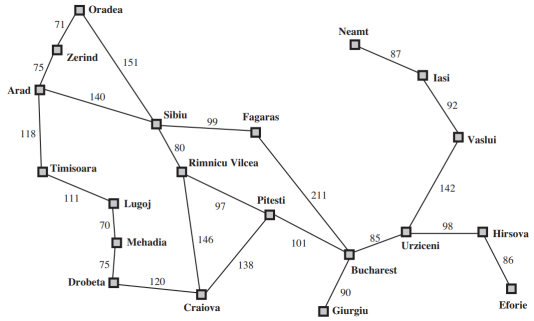

Figura 4. Mapa simplificado de uma reigão da Romênia, com a distância rodoviárias em milhas. Fonte: Russell and Norvig [2010]

Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

Figura 5. Lista de cidades com a distância estimada em linha reta até *Bucharest*. Fonte: Russell and Norvig [2010]

B. Busca Gulosa

Para implementar o algoritmo de busca gulosa, em C++, utilizamos duas estruturas de dados: o mapa [2] e a lista. O mapa é uma estrutura que associa um valor a uma chave, enquanto a lista é uma sequência de elementos do mesmo tipo. Para modelar o grafo, criamos uma estrutura chamada ”vizinhos”, que contém a cidade vizinha e a distância até essa cidade. Em seguida, criamos um mapa que associa cada cidade a uma lista de vizinhos. Por fim, criamos uma lista para armazenar as cidades já percorridas no caminho. Essa lista é necessária para guardar o todo o percurso e será impressa no final da execução do código.

Listing 1. Estruturas e variáveis.

```
struct vizinhos{
  string cidade;
  int dist;
};

string inicio, fim;
vizinhos aux;
map<string, int> distBucharest;
map<string, list<vizinhos>> mapa;
list<string> percorridos;

inicio = "Arad";
fim = "Bucharest";
```

Para implementar o algoritmo de caminho mais curto, utilizamos o a lógica explicada na subseção *Problema*. Esse algoritmo começa pelo vértice de origem e caminha para um dos vértices adjacentes a ele, sempre considerando como refência a distância da próxima cidade para Bucharest. Em seguida, ele armazena a cidade visitada na lista de ”percorridos”e repete o processo até que a cidade de destino seja alcançada. No nosso exemplo, a cidade de origem é ”Arad”e a cidade de destino é sempre ”Bucharest”. O algoritmo começa guardando a distância (aresta) de Arad até a cidade vizinha mais próxima de Bucharest e faz isso com todas até o final da execução, momento onde a distância percorrida pela rota será apresentada para o usuário.

Listing 2. Laço principal da Busca Gulosa.

```
string percorrido;
int distTotal = 0, dist = 0;
percorridos.push_back(inicio);

while(inicio != fim){

  int menorDistBucharest = distBucharest[
      mapa[inicio].front().cidade];
  percorrido = mapa[inicio].front().
      cidade;
  dist = mapa[inicio].front().dist;
  mapa[inicio].pop_front();

  while(!mapa[inicio].empty()){
    if(distBucharest[mapa[inicio].front()
        .cidade] < menorDistBucharest){
      menorDistBucharest = distBucharest[
          mapa[inicio].front().cidade];
      percorrido = mapa[inicio].front().
          cidade;
      dist = mapa[inicio].front().dist;
    }
    mapa[inicio].pop_front();
  }
  percorridos.push_back(percorrido);
  inicio = percorrido;
  distTotal += dist;
}
```

*C. Busca A**

A principal mudança do algoritmo A* em relação a busca gulosa é o critério para escolher a próxima cidade (vértice), desta vez além da distância em linha reta para Bucharest, também é considerada a distância local (aresta) entre as cidades, desta forma a próxima cidade a ser visitada é sempre a que tiver a menor distância somada destes dois valores. Este algoritmo é capaz de gerar percursos menores, entretanto alguns cuidados devem ser tomados, por exemplo a possibilidade da melhor escolha ser uma cidade já visitada. Para corrigir isso, foi adicionado ao código uma função de busca que faz uso da lista ”percorridos”onde estão listadas todas as cidades já visitadas, caso a busca retorne verdadeiro a cidade em questão será desconsiderada.

Listing 3. Função de busca na lista de percorridos.

```
bool busca(list<string> lista, string item)
    {
```

```
    list<string> aux = lista;
    while(!aux.empty()){
      if(item == aux.front()){
        return true;
      }
      else
        aux.pop_front();
    }
    return false;
}
```

Listing 4. Laço principal da Busca A*.

```
    string percorrido;
    int distTotal = 0, dist = 0;
    percorridos.push_back(inicio);

    while(inicio != fim){

      int menorDistBucharest = INT_MAX;
      int distancia = 0;
      int tam = mapa[inicio].size();

      percorrido = mapa[inicio].front().
          cidade;
      dist = mapa[inicio].front().dist;

      for(int i = 0; (i < tam) && !(mapa[
          inicio].empty()); i++) {

        distancia = (distBucharest[mapa[
            inicio].front().cidade] + mapa[
            inicio].front().dist);

        if((distancia < menorDistBucharest)
            && !(busca(percorridos, mapa[
            inicio].front().cidade))){
          menorDistBucharest = distancia;
          percorrido = mapa[inicio].front().
              cidade;
          dist = mapa[inicio].front().dist;
        }
        mapa[inicio].pop_front();
      }
      percorridos.push_back(percorrido);
      inicio = percorrido;
      distTotal += dist;
    }
```

D. Resultados

Pode-se verificar nos testes realizados que, o algoritmo A* se sobressai em cidades nas quais possuem mais de uma rota possível até *Bucharest*.

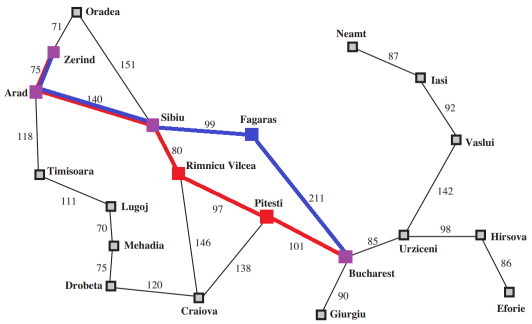

Figura 6. Melhores rotas encontradas pelos dois algoritmos de busca a Gulosa (em azul) e A* (em vermelho) de *Zerind* até *Bucharest*. Fonte: Russell and Norvig [2010] *Obs:* As rotas em destaque foram confeccionadas através de um software edição.

Como podemos observar na Figura 6, a melhor rota de *Zerind* até *Bucharest* foi obtida do algoritmo A*, que retornou uma distância de 493 *milhas*, contra 525 *milhas* da busca gulosa. A diferença na resposta se dá quando ambos algoritmos ”passam”pela cidade de *Sibiu*, pois nessa cidade o algoritmo ”guloso”opta por ir para **Fagaras** que está mais perto de **Bucharest**, porém o trajeto de *Sibiu* até ela é maior, enquanto o algoritmo A* decide ir por *Rimnicu Vilcea*.

Entretando, em cidades que possuem uma ou duas rotas (na qual há um distância consideravel entre elas) tanto a busca gulosa quanto a A* apresentam o mesmo resultado, como podemos observar na tabela II.

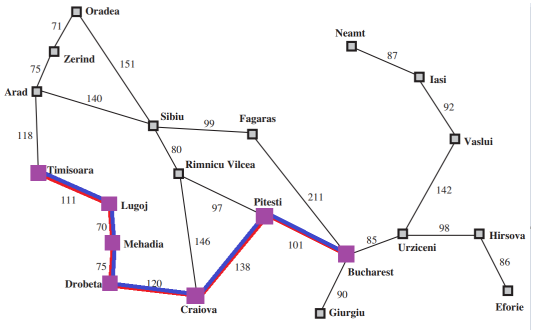

Figura 7. Melhores rotas encontradas pelos dois algoritmos de busca a Gulosa (em azul) e A* (em vermelho) de *Timisoara* até *Bucharest*. Fonte: Russell and Norvig [2010] *Obs:* As rotas em destaque foram confeccionadas através de um software edição.

Conforme ilustrado na figura 7, podemos notar que ambas as buscas resultaram na mesma resposta (615 *milhas*). Essa situação se repetiu nas cidades *Neamt*, *Iasi*, *Vaslui* e *Hirsova*, pois estas cidades apresentam um único caminho até *Bucharest*.

Com isso, podemos diz quer a busca a-estrela apresenta uma ligeira diferença em comparação com a busca gulosa, como podemos observar na tabela II.

Tabela II
DISTÂNCIA EM MILHAS OBTIDA DA MELHOR ROTA DE CADA BUSCA

Cidade	Busca Gulosa	Busca A*	Menor rota possível
Neamt	406	406	406
Timisoara	615	615	536
Fagaras	211	211	211
Arad	450	418	418
Oradea	461	429	429
Zerind	525	493	493

Porém, nas abordagens utilizadas nem sempre a solução encontrada será a menor possível. Pois quando a cidade inicial é *Timisoara* as respostas das duas buscas não foram a melhor possível. Depois de realizar uma análise nas rotas a partir da cidade citada anteriormente, podemos observar que a rota ótima seria obtida indo para *Arad* ao invés de *Lugoj*, esse caminho teria um custo maior a princípio, entretando este trajeto seria mais curto até chegar a *Bucharest*, pois a distância final seria igual a 536 *milhas*.

E. Conclusão

Nesta atividade foram implementados dois algoritmos de busca: Gulosa e A*, visando a resolução do problema de otimização de rota proposta pelo Russel e Norvig e comparamos as respostas obtidas pelas buscas citadas anteriormente.

Podemos concluir que o melhor uso da busca gulosa é quando não temos informações precisas sobre o caminho para o objetivo, mas sabemos qual é o objetivo final. Já a busca A* é usada quando temos informações precisas sobre o caminho para o objetivo, ou seja, quando temos uma heurística que nos permite estimar a distância ou custo para alcançar o objetivo.

Nenhuma das duas buscas garante a melhor solução sempre, para isso seria necessária a implementação da busca de custo uniforme, pois esta expande todos os caminhos e verifica todas as possibilidades, assim, apresentando o sempre a melhor rota (desde que o espaço de busca seja finito), porém a um custo bem maior.

REFERÊNCIAS

[1] Russell, S. & Norvig, P. (2010). *Artificial Intelligence: A Modern Approach*. 3rd ed. Upper Saddle River, NJ: Pearson.

[2] GEEKSFORGEEKS. *Map, Associative Containers - The C++ Standard Template Library (STL)*. Disponível em: <https://www.geeksforgeeks.org/map-associative-containers-the-c-standard-template-library-stl/>. Acesso em: 29 abr. 2023.