UNIVERSIDADE FEDERAL DE SANTA CATARINA
INTELIGÊNCIA ARTIFICIAL APLICADA A AUTOMAÇÃO

BRUNO SOUZA DE LIMA

IMPLEMENTAÇÃO DE ALGORITMOS DE BUSCA PARA SOLUÇÃO DO PROBLEMA "ELEVATOR LOGICS"

FLORIANÓPOLIS
2019

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SÍMBOLOS

BL	Busca em largura
BP	Busca em profundidade
BPI	Busca em profundidade iterativa
b	Fator de ramificação
d	Profundidade da solução mais próxima à raiz
l	Profundidade limite da árvore
m	Profundidade máxima da árvore
$f(n)$	Função estimativa
$h(n)$	Função heurística
$g(n)$	Função menor custo de caminho

SUMÁRIO

1 INTRODUÇÃO

A inteligência artificial (IA) tem revolucionado diversos setores da indústria e do mercado, trazendo maior velocidade a processos e maior certeza em tomada de decisões. Podemos ver exemplos de inteligência artificial em *chatbots*, análises de risco, sistemas de recomendação, carros autônomos, aplicativos de rotas, *business intelligence*, reconhecimento facial, entre muitos outros.

O interesse e investimento em IA tem crescido muito nos últimos anos, "*Big bang of modern AI*"[1] ocorrido no começo da década, possibilitado pela união de três fatores: a gigantesca quantidade de dados produzida pela humanidade (*Big Data*), os modelos de *Machine Learning* que existem atualmente e a super computação acessível. Segundo a NVIDIA, o número de estudantes da Udacity em cursos de IA aumento em cem vezes em dois anos (2015 a 2017), e o investimento em *Startups* de IA aumentou nove vezes em quatro anos (2012 a 2016), chegando a cinco bilhões de dólares.

Uma forma de resolução de problemas de IA há muito tempo já difundida e utilizada é por meio de busca. A busca tenta encontrar um caminho entre um estado inicial e um estado meta, e esse caminho é um conjunto de ações que alteram o estado inicial, transformando-o em estados intermediários, até se assemelhar ao estado meta. Por exemplo, um aplicativo de GPS tentando encontrar o melhor caminho para o usuário ir da sua casa para o trabalho.

O objetivo deste trabalho é implementar e comparar diferentes algoritmos de busca para a resolução do problema "*Elevator logics*". O trabalho é dividido em quatro partes, primeiro é feita uma descrição do problema a ser resolvido. Na segunda parte são descritos quais são as formas de busca que serão implementadas, e decidindo se são aplicáveis ao problema. A terceira parte é a implementação, sendo discutidas otimizações e as heurísticas utilizadas, e na última estão os resultados.

2 DESCRIÇÃO DO PROBLEMA

2.1 Descrição informal

O problema "*Elevator logics*" é um problema de lógica, usado como um jogo para testar seu raciocínio ou apenas entreter, e pode ser acessado pelo link *Elevator logics*.

O objetivo do problema é abrir os 5 elevadores, cada um com uma pessoa dentro, liberando-as. Porém os elevadores só se abrem se estiverem todos parados em algum andar entre o 21 e o 25, no nível 1 do jogo, ou entre o 21 e o 23, no nível 2. No nível 1 os elevadores começam nos andares 17, 26, 20, 19 e 31. No nível 2 eles estão inicialmente nos andares 20, 20, 22, 24 e 21.

Os elevadores ainda tem uma restrição de movimento, eles só podem subir 8 andares de uma vez, ou descer 13 andares de uma vez. Eles podem realizar essas duas ações quantas vezes quiserem, porém 2, e apenas 2, elevadores devem realizar a ação simultaneamente.

O prédio possui 50 andares (numerados de 0 a 49), então se um elevador está em um andar acima do 41, ele não pode realizar a ação de subir pois iria para o andar 50, que não existe. Da mesma forma, um elevador no andar 12 ou abaixo não pode descer.

O problema não impõe restrição de quantidade de ações, porém nesse trabalho uma solução otimizada será uma solução com o menor número de ações possível.

2.2 Descrição formal

Um problema pode ser definido formalmente pelos seguintes componentes[2]:

- O **estado inicial** do problema.
- As **ações** executáveis no estado atual.
- O **modelo de transição**, que dita como as ações alteram o estado.
- O **teste de meta**, que avalia se o estado atual é um estado meta.
- O **custo do caminho**, que define um custo para uma determinada ação.

Traduzindo a descrição informal do problema nos componentes citados, e adicionando um componente **estados**, pode-se descrever o problema.

- **Estados:** o estado do problema é determinado pelo número do andar de cada elevador. Como são 5 elevadores, e estimando que cada elevador pode estar em qualquer um dos 50 andares, há um total de 50^5 estados possíveis, ou 312.500.000.
- **Estado inicial:** os elevadores nos andares 17, 26, 20, 19 e 31, no nível 1, e nos andares 20, 20, 22, 24 e 21, no nível 2 (Figura 1).

- **Ações:** dois elevadores subirem 8 andares ou dois elevadores descerem 13 andares. Há a restrição do número de andares, não é possível realizar a ação de subir em um elevador no andar 42 ou acima, nem realizar a ação de descer em um elevador no andar 12 ou abaixo.
- **Modelo de transição:** altera o número do andar do elevador que sofre a ação. Se o elevador sobe 8 andares é somado 8 ao valor do seu andar, se o elevador desce 13 andares é subtraído 13 do valor do seu andar.
- **Teste de meta:** checa se todos os elevadores estão em um andar entre 21 e 25, inclusos, no nível 1, ou entre 21 e 23, inclusos, no nível 2 (Figura 2).
- **Custo do caminho:** cada ação custa 1, então o custo do caminho é igual ao número de ações necessárias para resolver o problema.

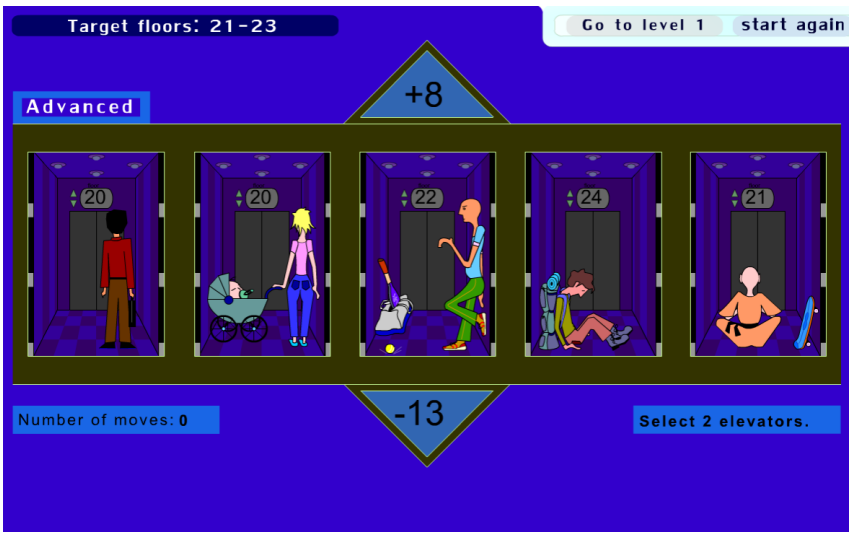

Figura 1 – Estado inicial. Fonte: Plastelina logic games.

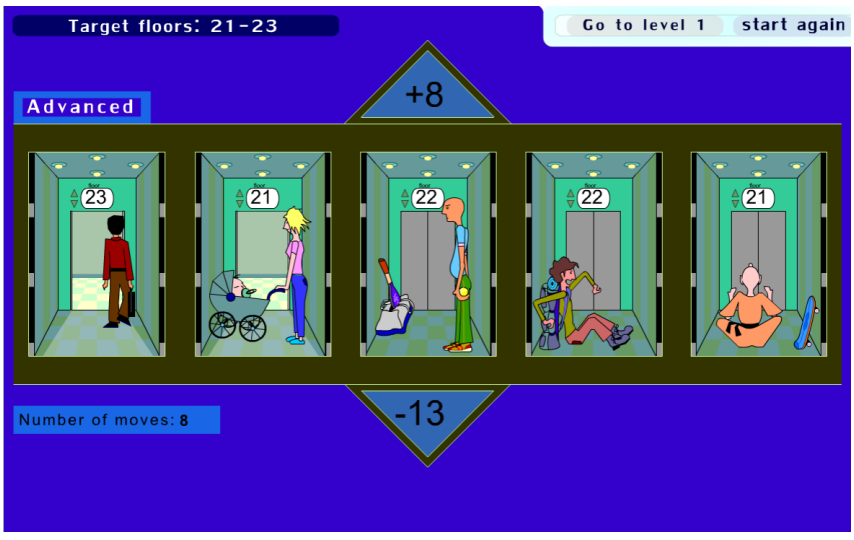

Figura 2 – Possível estado meta. Fonte: Plastelina logic games.

3 ALGORITMOS DE BUSCA

Tendo uma descrição formal do problema, o próximo passo é encontrar uma solução. A solução do problema é o caminho, a sequência de ações, que transforma o estado inicial no estado meta. A função dos algoritmos de busca é considerar várias sequências de ações possíveis, e retornar o(s) caminho(s) que levam ao estado meta, podendo retornar, ou não, o caminho mais otimizado segundo algum critério.

Para uma melhor compreensão do funcionamento de algoritmos de busca, utiliza-se o conceito de árvore de busca. Nessa representação, o estado inicial representa a raiz da árvore, os galhos representam as ações e os nodos representam os estados. A Figura 3 retrata o começo de uma árvore de busca, muito simplificada, do problema. Cada conjunto de cinco retângulos simboliza os cinco elevadores com o número de seu andar no centro.

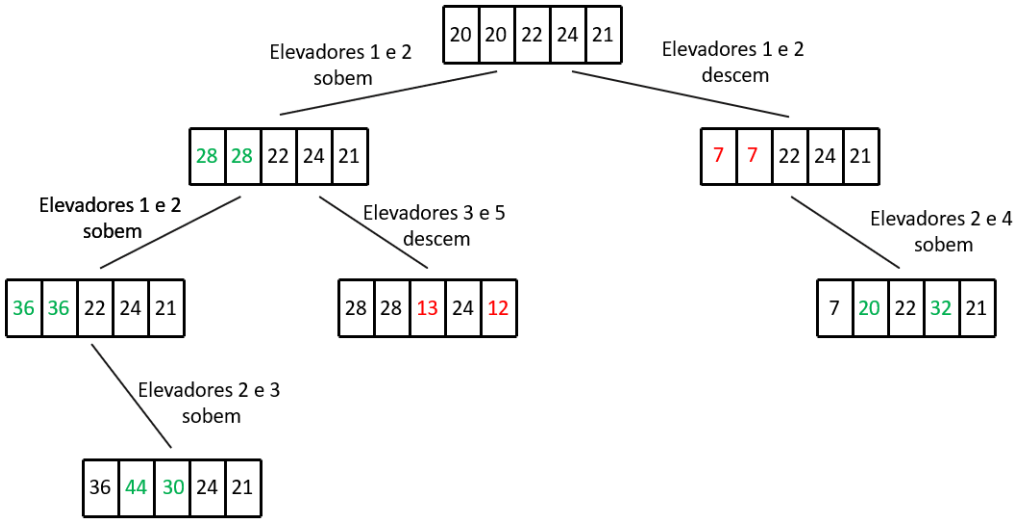

Figura 3 – Árvore de busca. Fonte: do autor

Nessa árvore foram escolhidas ações aleatórias para fins didáticos. Na árvore real do problema, partindo do estado inicial, há vinte possíveis ações, subir ou descer as dez combinações diferentes dos cinco elevadores, dois a dois.

Dizemos que o galho parte do nodo pai e vai para um nodo sucessor, os nodos que não possuem sucessores são chamados de folhas (da árvore). Então do estado inicial foram gerados dois nodos sucessores. É necessário, então, escolher qual dos nodos será expandido em seguida, o conjunto de nodos disponíveis para serem expandidos é chamado de fronteira.

Os nodos são expandidos até se encontrar um estado meta, ou não haver mais nodos para expandir. O que diferencia os algoritmos de busca é como os nodos a serem expandidos são escolhidos, qual a sua estratégia de busca.

3.1 Estrutura

A estrutura de cada nodo da árvore de busca em um algoritmo pode ser definida com quatro componentes[2]:

- O **estado** correspondente ao nodo.
- O nodo **pai**, gerador do nodo atual.
- A **ação** utilizada no nodo pai para gerar o nodo atual.
- O **custo do caminho** da raiz até o nodo atual.

3.2 Desempenho

A estratégia de busca pode ser escolhida se baseando em um ou mais de quatro critérios[2]:

- **Completeza:** Um algoritmo de busca é completo se, dado que haja solução para o problema, ele consegue encontrá-la.
- **Ótimo:** O algoritmo é ótimo se consegue encontrar a melhor solução para o problema.
- **Tempo:** O tempo necessário para encontrar a solução.
- **Espaço:** A quantidade de memória necessária para encontrar a solução.

3.3 Busca não informada

Algoritmos de busca não informada não tem conhecimento para julgar se um estado é melhor que outro, se o caminho que estão seguindo é o melhor caminho. Eles só conseguem diferenciar se o estado atual é o estado meta ou não. O que distingue esse tipo de algoritmo é a ordem em que os nodos serão expandidos.

3.3.1 Busca por largura

Na busca por largura, após o nodo raiz ser expandido, todos os seus sucessores são expandidos. Após todos os sucessores serem expandidos, os sucessores desses sucessores são expandidos.

Diz-se que a fronteira é uma fila, ou FIFO (*first in, first out*), ou seja, o primeiro que entra na fila é o primeiro a sair. Um exemplo comum é uma fila de supermercado, o primeiro a ser atendido é o primeiro que entrou na fila, as pessoas vão sendo atendidas por ordem de chegada.

A Figura 4 mostra a ordem de expansão para a árvore da Figura 3.

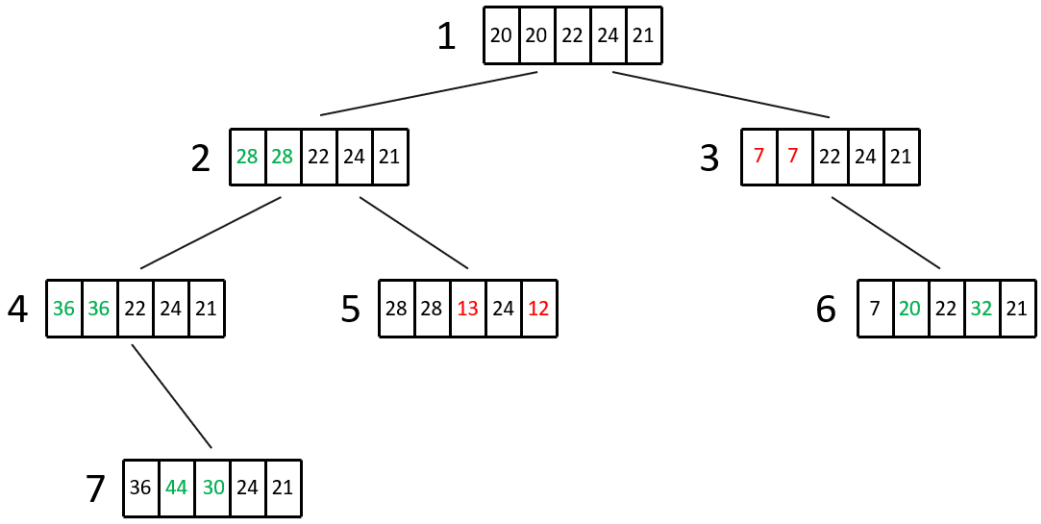

Figura 4 – Busca por largura. Fonte: do autor

O nodo raiz (1) é expandido e seus sucessores (2 e 3) entram na fila. O nodo 2 é expandido e seus sucessores (4 e 5) entram na fila após o nodo 3. Então o nodo 3 é expandido e seu sucessor (6) entra na fila após o nodo 5. Finalmente o nodo 4 é expandido e seu sucessor (7) é o último da fila. Essa busca é chamada de busca por largura pois são expandidos todos os nodos de um certo nível (largura) da árvore, para só depois ir para um nível abaixo.

Essa estratégia é completa pois percorre todos os nodos da árvore até encontrar a solução. É ótima, no problema dos elevadores, porque encontra a solução mais próxima à raiz, ou seja, a com menor número de ações. O tempo e o espaço necessários para a solução são proporcionais à ordem de grandeza do fator de ramificação b elevado à profundidade da solução d (em que nível da árvore ela está), escrevemos $O(b^d)$.

3.3.2 Busca de custo uniforme

Nessa forma de busca, a ordem da fila é pelo custo do caminho do nodo, estimado por $g(n)$, do menor para o maior. Essa estratégia é boa quando há custos diferentes para cada sucessor de um nodo. Como o custo de um pai para um sucessor no problema apresentado é sempre 1, essa busca se tornaria uma busca em largura. Por essa razão a busca de custo uniforme não será implementada.

3.3.3 Busca por profundidade

Na busca por profundidade o algoritmo, após expandir a raiz, expande seu primeiro sucessor, então expande o primeiro sucessor do primeiro sucessor e continua assim até a maior profundidade possível, em que o nodo não possui sucessores.

A fronteira nesse caso é uma pilha, ou LIFO (*last in, first out*), em que o último a entrar é o primeiro a sair. Como exemplo pode-se pensar em uma pilha de caixas. Há uma

caixa vermelha no chão e uma pessoa empilha uma caixa azul em cima, depois empilha uma caixa verde e por último uma caixa amarela. Quando a pessoa for pegar as caixas de volta, ela vai começar pela amarela, depois verde, azul e finalmente a vermelha. Ela pega primeiro a última caixa que foi colocada na pilha.

A ordem dos nodos na busca por profundidade na árvore exemplo é apresentada na Figura 5.

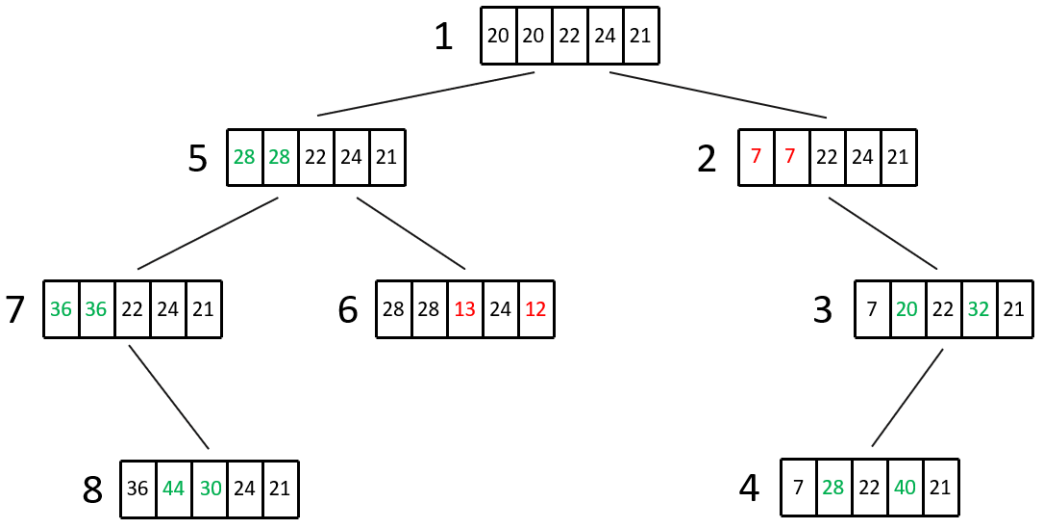

Figura 5 – Busca por profundidade. Fonte: do autor

Ao expandir a raiz, são gerados dois sucessores, é expandido então o último sucessor (2), gerando o nodo 3, q é expandido gerando o sucessor 4. Como o nodo 4 não possui sucessores, retorna-se ao 3. O único sucessor do nodo 3 é o 4 que já foi percorrido, então volta-se ao nodo 2, porém todos os sucessores do nodo 2 também já foram visitados. É expandido, agora, o outro sucessor da raiz (5), gerando 6 e 7. O algoritmo escolhe primeiro a direção de 6, pois foi o último nodo gerado, depois segue o caminho do 7.

Essa estratégia de busca é completa, se não há repetição de estados, ou seja, se um estado que já foi testado não precisar ser testado novamente. Se houver repetição de estados a busca pode entrar em *loop*. No caso desse trabalho não haverá repetição de estado. A estratégia não é ótima, pois pode encontrar uma solução com número de ações maiores. O tempo necessário para encontrar a solução é da ordem $O(b^m)$, em que m é a profundidade máxima de qualquer nodo. A vantagem da busca por profundidade é a memória utilizada. Se um nodo já teve toda sua linhagem de sucessores explorada e não se encontrou uma solução, é possível removê-lo da memória, junto com seus sucessores. A memória utilizada então é da ordem de grandeza $O(b.m)$.

3.3.4 Busca por profundidade limitada

Essa estratégia de busca é semelhante à busca por profundidade. A diferença que existe é que existe um limite de profundidade l. Ao chegar nessa profundidade o

algoritmo não expande os nodos e tenta outro caminho. A Figura 6 mostra a ordem que serão percorridos os estados se a profundidade máxima l for 2 (considerando que a raiz tem profundidade 0).

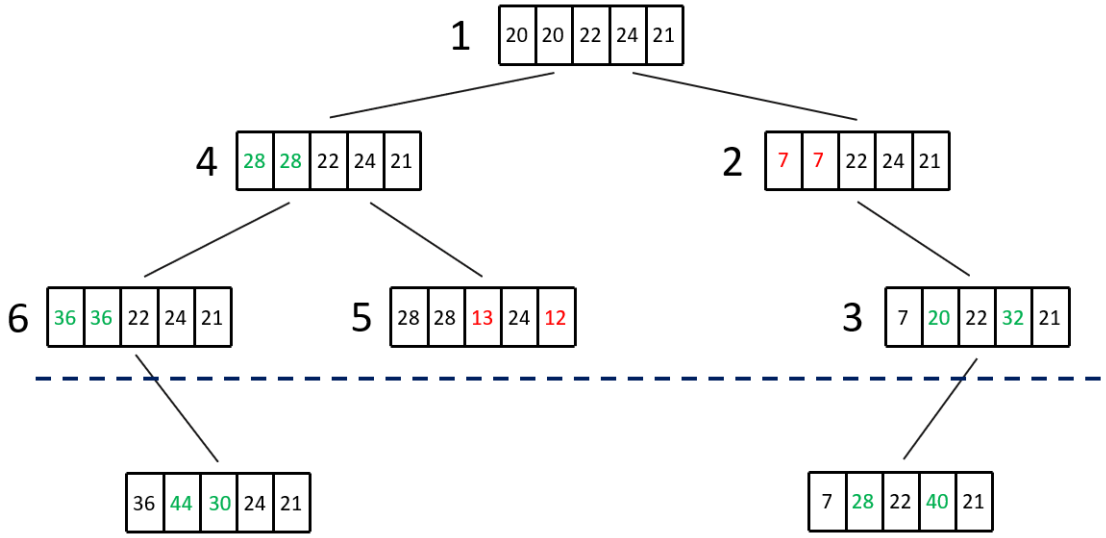

Figura 6 – Busca por profundidade limitada. Fonte: do autor

Essa estratégia não será implementada, pois não sabemos em que profundidade está a solução, não sabemos o número de ações necessárias. Sendo assim é muito difícil saber se a profundidade escolhida está sub ou superestimada, esse tipo algoritmo é bom quando temos certeza da profundidade da solução.

3.3.5 Busca por profundidade iterativa

Esse algoritmo também trabalha com uma profundidade limite. Porém cada vez que ele tenta resolver o problema, se não for encontrada uma solução, o algoritmo incrementa uma unidade na profundidade limite.

Essa estratégia é completa, pois o código incrementará a profundidade limite até encontrar uma solução. É também ótima, no nosso caso, pois encontra a solução mais próxima à raiz. O tempo é relacionado à ordem de grandeza da última árvore testada, $O(b^d)$. O a memória necessária tem relação com a ordem de grandeza $O(b.d)$.

3.3.6 Busca bidirecional

A busca bidirecional atua como duas buscas (podendo-se escolher entre as estratégias anteriores). Uma das buscas começa da raiz e vai em direção à solução, enquanto a outra busca começa da solução e vai em direção à raiz. Quando as duas buscas encontram um mesmo estado, a busca acaba, pois é possível traçar um caminho do estado inicial ao estado meta. A Figura 7 mostra a ordem em que os nodos são percorridos em uma busca bidirecional.

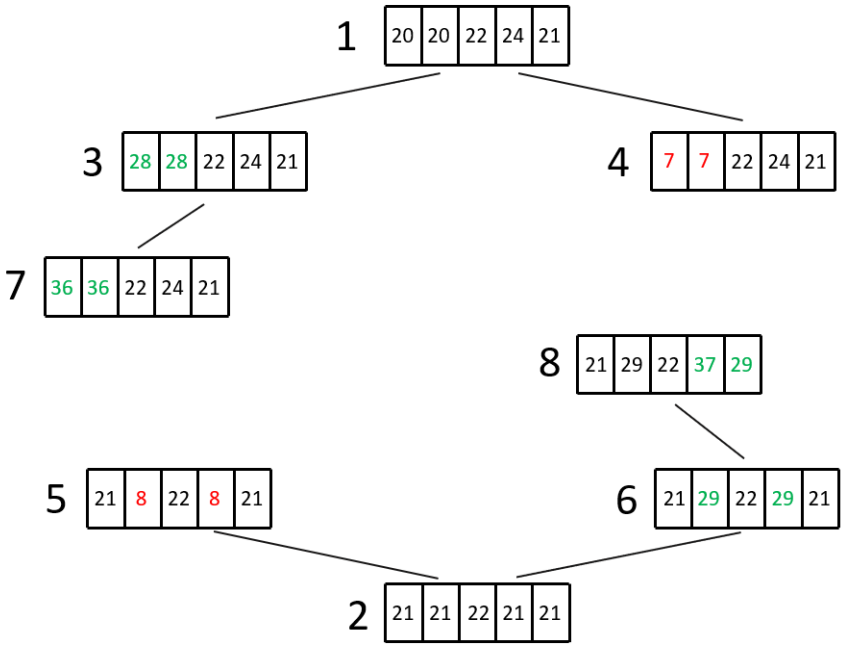

Figura 7 – Busca bidirecional. Fonte: do autor

Essa estratégia também não será implementada. Ela é boa quando o estado meta é conhecido, porém há 3.125 opções diferentes no nível 1, e 243 opções de estado meta no nível 2. Não é possível saber qual estado é uma opção melhor.

3.3.7 Comparação

Os algoritmos de busca não informada a serem implementados são: busca por largura, busca por profundidade e busca por profundidade iterativa. A Tabela 1 apresenta uma comparação entre os três. Busca de custo uniforme não será implementada pois se tornaria uma busca por largura nesse problema. A busca por profundidade limitada e busca bidirecional necessitam de informações mais detalhadas sobre o estado meta.

Tabela 1 – Comparação entre estratégias de busca não informada

	BL	BP	BPI
Completa	Sim	Sim*	Sim*
Ótima	Sim	Não	Não
Tempo	$O(b^d)$	$O(b^m)$	$O(b^d)$
Espaço	$O(b^d)$	$O(b.m)$	$O(b.d)$

*Completa se a busca for sem repetição de estados.
Fonte: do autor.

3.4 Busca informada

Na busca informada o algoritmo é capaz de julgar se um estado é melhor que outro por meio por meio de uma função estimativa $f(n)$. A ordem de expansão dos nodos

é do nodo que tem menor valor para $f(n)$ para o que tem maior valor. A diferença entre os algoritmos de busca informada é a forma que a função $f(n)$ é construída.

3.4.1 Busca gulosa

Na busca gulosa, a função estimativa é igual à função heurística ($f(n) = h(n)$). O algoritmo sempre vai priorizar o nodo mais perto do objetivo sem levar em conta seu custo. Esse método não será implementado, pois o foco da busca informada será a busca A*, que pode ser encarada como uma versão aprimorada da busca gulosa, pois leva em conta também o custo do caminho.

3.4.2 Busca A*

A busca A* usa a função estimativa como a soma do custo do caminho e a função heurística ($f(n) = g(n) + h(n)$), que pode ser visto como o custo da solução, sempre expandindo o nodo com menor valor de $f(n)$ primeiro. A busca de custo uniforme (Seção 3.3.2) é um caso especial da função A* em que $h(n) = 0$.

Para que a busca A* seja ótima, a heurística $h(n)$ deve ser admissível. Para que $h(n)$, seja admissível, $h(n) \leq h^*(n)$, em que h^* é o custo real de n até a meta[3].

O método A* é completo e ótimo, se $h(n)$ é admissível. No pior dos casos o tempo e o espaço necessários são relacionados com $O(b^d)$.

As heurísticas utilizadas são apresentadas na Seção 4.2.

4 IMPLEMENTAÇÃO

Os códigos foram escritos na linguagem Python no Visual Studio Code e estão disponíveis no GitHub através do link *GitHub Bruno*.

Todos os códigos foram desenvolvidos pelo autor, exceto PriorityQueue.py. A implementação da *priority queue* foi extraída do código utils.py disponível no link *GitHub AIMA python*[4]. As estratégias implementadas foram busca por largura, profundidade, profundidade iterativa e A*, todas sem repetição de estados.

4.1 Otimização

4.1.1 Lista e *set*

Para não haver repetição de estados, os estados já testados são armazenados em uma estrutura de dados. Cada nodo antes de ser testado, tem seu estado verificado. Se é um estado que já está contido na estrutura de dados, esse nodo é ignorado.

Primeiramente a implementação das buscas foi feita utilizando uma lista para armazenar esses estados, o que fazia com que o código levasse muito tempo para encontrar uma solução. Para checar se um item está contido em uma lista, a ordem de grandeza é $O(n)$ e em um *set* é $O(1)$[5].

A tabela 2 compara a velocidade de execução para busca em largura, profundidade, profundidade iterativa e A* utilizando a heurística 2, para o problema no nível 2. Com base nos resultados foi alterada a forma de armazenar estados testados para um *set*.

	BL	BP	BPI	A* (h2)
Tempo com lista	*	*	22min 1s	2h 59min 23s
Tempo com *set*	21,2s	*	18,3s	0,112s

* Erro de memória

Tabela 2 – Comparação do tempo de execução com lista e *set*

4.1.2 Lista e *priority queue*

Para as buscas A*, a *priority queue* foi primeiramente implementada como uma lista, e cada vez que um nodo era adicionado na lista era utilizada a função sort() para ordernar os nodos.

A utilização da lista foi comparada na Tabela 3 com o uso de uma *priority queue* pronta feita com a biblioteca *heapq*. A heurística utilizada para a comparação foi a heurística 1 no nível 1 do problema.

	Lista	*Priority queue*
Tempo de execução (s)	30,56	1,34

Tabela 3 – Comparação performance lista e *priority queue*

O algoritmo com a *priority queue* encontoru a solução muito mais rápido, pois a complexidade temporal do método *heappush()* da biblioteca *heapq* possui ordem de grandeza $O(log\ n)$[6], enquanto a ordem de grandeza para o método *sort()* é de $O(n\ log\ n)$[5]. Então optou-se por utilizar a *priority queue* em vez de lista.

4.2 Heurísticas

A seguir são apresentadas as heurísticas implementadas.

4.2.1 Heurística 1

A heurística 1 (Figura 8) simplesmente retorna o número de elevadores que não está dentro da faixa de andares da meta.

```
def Heuristica1(estado, meta):
    distancia = 0
    for andar in estado:
        if andar not in meta:
            distancia += 1
    return distancia
```

Figura 8 – Heurística 1 (h1). Fonte: do autor

4.2.2 Heurística 2

A heurística 2 (Figura 9) é um pouco mais complexa que a heurística anterior. Se um elevador só precisa de uma ação para chegar na meta ele tem valor 1, se ele necessita de duas ações ele tem valor 2. Caso o elevador precise de mais de duas ações para chegar na meta ele tem valor 3. A heurística retorna a soma dos valores dos 5 elevadores

```python
def Heuristica2(estado, meta):
    distancia = 0
    for andar in estado:
        if andar in meta:
            pass
        elif (andar + 8 in meta) or (andar - 13 in meta):
            distancia += 1
        elif (andar + 16 in meta) or (andar - 26 in meta) or (andar - 5 in meta):
            distancia += 2
        else:
            distancia += 3
    return distancia
```

Figura 9 – Heurística 2 (h2). Fonte: do autor

4.2.3 Heurística 3

A heurística 3 (Figura 10) é semelhante à heurística 2, porém se um elevador necessita de três ações para chegar à meta ele tem valor 3. E mais de três ações tem valor 4.

```python
def Heuristica3(estado, meta):
    distancia = 0
    for andar in estado:
        if andar in meta:
            pass
        elif (andar + 8 in meta) or (andar - 13 in meta):
            distancia += 1
        elif (andar + 16 in meta) or (andar - 26 in meta) or (andar - 5 in meta):
            distancia += 2
        elif (andar + 24 in meta) or (andar - 39 in meta) or (andar - 18 in meta) or \
            (andar + 3 in meta):
            distancia += 3
        else:
            distancia += 4
    return distancia
```

Figura 10 – Heurística 3 (h3). Fonte: do autor

4.2.4 Heurística 4

A heurística 4 (Figura 11) é semelhante à heurística 3, porém se um elevador necessita de quatro ações para chegar à meta ele tem valor 4. E mais de quatro ações tem valor 5.

```python
def Heuristica4(estado, meta):
    distancia = 0
    for andar in estado:
        if andar in meta:
            pass
        elif (andar + 8 in meta) or (andar - 13 in meta):
            distancia += 1
        elif (andar + 16 in meta) or (andar - 26 in meta) or (andar - 5 in meta):
            distancia += 2
        elif (andar + 24 in meta) or (andar - 39 in meta) or (andar - 18 in meta) or \
            (andar + 3 in meta):
            distancia += 3
        elif (andar + 32 in meta) or (andar + 11 in meta) or (andar - 31 in meta) or \
            (andar - 10 in meta):
            distancia += 4
        else:
            distancia += 5
    return distancia
```

Figura 11 – Heurística 4 (h4). Fonte: do autor

4.2.5 Heurística 5

A heurística 5 (Figura 12) é semelhante à heurística 4, porém se um elevador necessita de cinco ações para chegar à meta ele tem valor 5. E mais de cinco ações tem valor 6.

```
def Heuristica5(estado, meta):
    distancia = 0
    for andar in estado:
        if andar in meta:
            pass
        elif (andar + 8 in meta) or (andar - 13 in meta):
            distancia += 1
        elif (andar + 16 in meta) or (andar - 26 in meta) or (andar - 5 in meta):
            distancia += 2
        elif (andar + 24 in meta) or (andar - 39 in meta) or (andar - 18 in meta) or \
            (andar + 3 in meta):
            distancia += 3
        elif (andar + 32 in meta) or (andar + 11 in meta) or (andar - 31 in meta) or \
            (andar - 10 in meta):
            distancia += 4
        elif (andar + 40 in meta) or (andar - 19 in meta) or (andar - 44 in meta) or \
            (andar - 2 in meta) or (andar - 23 in meta):
            distancia += 5
        else:
            distancia +=6
    return distancia
```

Figura 12 – Heurística 5 (h5). Fonte: do autor

4.2.6 Heurística 6

Por último, a heurística 6 (Figura 13) é semelhante à heurística 5, porém se um elevador necessita de seis ações para chegar à meta ele tem valor 6. E mais de seis ações tem valor 7.

```python
def Heuristica6(estado, meta):
    distancia = 0
    for andar in estado:
        if andar in meta:
            pass
        elif (andar + 8 in meta) or (andar - 13 in meta):
            distancia += 1
        elif (andar + 16 in meta) or (andar - 26 in meta) or (andar - 5 in meta):
            distancia += 2
        elif (andar + 24 in meta) or (andar - 39 in meta) or (andar - 18 in meta) or \
            (andar + 3 in meta):
            distancia += 3
        elif (andar + 32 in meta) or (andar + 11 in meta) or (andar - 31 in meta) or \
            (andar - 10 in meta):
            distancia += 4
        elif (andar + 40 in meta) or (andar - 19 in meta) or (andar - 44 in meta) or \
            (andar - 2 in meta) or (andar - 23 in meta):
            distancia += 5
        elif (andar + 48 in meta) or (andar + 27 in meta) or (andar + 6 in meta) or \
            (andar - 36 in meta) or (andar - 15 in meta):
            distancia += 6
        else:
            distancia += 7
    return distancia
```

Figura 13 – Heurística 6 (h6). Fonte: do autor

5 RESULTADOS

A seguir são apresentados os resultados de cada busca implementada para cada nível. Embora o problema tenha apenas dois níveis, foi **acrescentado um nível 3, em que o estado inicial é o estado do nível 1 (andares 17, 26, 20, 19 e 31) e a meta é a meta do nível 2 (andares 21, 22 e 23)**.

5.1 Nível 1

Na Tabela 4 é possível ver que todas as buscas conseguiram encontrar solução, exceto a BP que apresentou um erro de memória. A partir da heurística 2 o número de nodos explorados caiu drasticamente, sendo apenas dezenas. A heurística 3 se mostrou a mais eficiente, explorando apenas 24 nodos em cerca de 0,03s. A heurística 4 não apresentou melhora em relação à heurística 3. Com base nisso decidiu-se otimizar a heurística 3.

	BL	BP	BPI	A*(h1)	A*(h2)	A*(h3)	A*(h4)
# nodos testados	562.923	*	5.503	13.699	36	24	24
Tempo de execução (s)	33,2	*	8,32	1,34	0,046	0,029	0,030

* Erro de memória

Tabela 4 – Resultados nível 1

A otimização da heurística 3 consistiu em alterar o valor retornado, dividindo-o por constantes. A maior otimização foi encontrada dividindo o valor retornado por 0,3 (Figura 14), então a heurística foi denominada heurística 3_03 (h3_03).

```python
def Heuristica3_03(estado, meta):
    distancia = 0
    for andar in estado:
        if andar in meta:
            pass
        elif (andar + 8 in meta) or (andar - 13 in meta):
            distancia += 1
        elif (andar + 16 in meta) or (andar - 26 in meta) or (andar - 5 in meta):
            distancia += 2
        elif (andar + 24 in meta) or (andar - 39 in meta) or (andar - 18 in meta) or \
            (andar + 3 in meta):
            distancia += 3
        else:
            distancia += 4
    return distancia/0.3
```

Figura 14 – Heurística 3_03 (h3_03). Fonte: do autor

A Tabela 5 compara a busca A* usando a melhor heurística com as outras buscas. A busca A* explorou 9 nodos, ou seja encontrou o caminho exato na primeira tentativa, o nodo inicial mais os 8 nodos gerados pelas 8 ações. Enquanto isso as outras buscas tiveram que explorar milhares ou centenas de milhares de nodos.

	BL	BP	BPI	A*(h3_03)
# nodos testados	562.923	*	5.503	9
Tempo de execução (s)	33,2	*	8,32	0,014

* Erro de memória

Tabela 5 – Resultados nível 1 com heurística h3_03

5.2 Nível 2

Os resultados do nível 2 foram semelhantes aos do nível 1, no sentido que a busca A* foi superior aos outros métodos, a diferença é que as heurísticas melhoraram até a heurística 5 (Tabela 6).

	BL	BP	BPI	A*(h1)	A*(h2)	A*(h3)	A*(h4)	A*(h5)	A*(h6)
# nodos testados	400.752	*	717.130	11.523	516	463	247	226	226
Tempo de execução (s)	21,2	*	18,3	1,53	0,112	0,107	0,080	0,072	0,077

* Erro de memória

Tabela 6 – Resultados nível 2

A otimização da heurística 5 foi feita dividindo o valor retornado por 0,3 (Figura 15. A nova heurística foi chamada de heurística 5_03 (h5_03).

```
def Heuristica5_03(estado, meta):
    distancia = 0
    for andar in estado:
        if andar in meta:
            pass
        elif (andar + 8 in meta) or (andar - 13 in meta):
            distancia += 1
        elif (andar + 16 in meta) or (andar - 26 in meta) or (andar - 5 in meta):
            distancia += 2
        elif (andar + 24 in meta) or (andar - 39 in meta) or (andar - 18 in meta) or \
            (andar + 3 in meta):
            distancia += 3
        elif (andar + 32 in meta) or (andar + 11 in meta) or (andar - 31 in meta) or \
            (andar - 10 in meta):
            distancia += 4
        elif (andar + 40 in meta) or (andar - 19 in meta) or (andar - 44 in meta) or \
            (andar - 2 in meta) or (andar - 23 in meta):
            distancia += 5
        else:
            distancia += 6
    return distancia/0.3
```

Figura 15 – Heurística 5_03 (h5_03). Fonte: do autor

A busca A* com a melhor heurística implementada conseguiu resolver o problema explorando apenas 182 nodos, enquanto as outras buscas necessitaram de centenas de milhares de nodos (Tabela 7).

	BL	BP	BPI	A*(h5_03)
# nodos testados	400.752	*	717.130	182
Tempo de execução (s)	21,2	*	18,3	0,067

* Erro de memória

Tabela 7 – Resultados nível 2 com heurística h5_03

5.3 Nível 3

No nível 3 (estado inicial do nível 1 com meta do nível 2), nenhuma das buscas não informadas conseguiu resolver o problema. A busca com a primeira heurística, mesmo sendo uma busca informada, necessitou explorar mais de um milhão de nodos, mostrando a complexidade do problema (Tabela 8).

A heurística 3 foi a que se saiu melhor nesse nível. Diferente dos outros niveis, em vez de as heurísticas posteriores apenas manterem o número de nodos explorados, nesse caso elas ficaram piores, aumentando esse número.

	BL	BP	BPI	A*(h1)	A*(h2)	A*(h3)	A*(h4)	A*(h5)
# nodos testados	**	*	*	1.082.324	110.930	74.816	78.752	98.897
Tempo de execução (s)	**	*	*	80,5	8.82	6,50	6,67	7,66

* Erro de memória
** Tempo muito longo

Tabela 8 – Resultados nível 3

Embora a heurística 3 tenha se mostrado superior à heurística 2, em termos de número de nodos explorados e tempo de execução, quando foram otimizadas, a heurística 2 modificada (Figura 16) se tornou mais eficiente que a heurística 3 otimizada. A nova heurística foi chamada de heurística 2_03 (h2_03).

```python
def Heuristica2_03(estado, meta):
    distancia = 0
    for andar in estado:
        if andar in meta:
            pass
        elif (andar + 8 in meta) or (andar - 13 in meta):
            distancia += 1
        elif (andar + 16 in meta) or (andar - 26 in meta) or (andar - 5 in meta):
            distancia += 2
        else:
            distancia += 3
    return distancia/0.3
```

Figura 16 – Heurística 2_03 (h2_03). Fonte: do autor

A busca A* com a heurística otimizada conseguiu achar a solução testando 889 nodos em 0,27 segundos (Figura 9).

	BL	BP	BPI	A*(h2_03)
# nodos testados	**	*	*	889
Tempo de execução (s)	**	*	*	0,27

* Erro de memória
** Tempo muito longo

Tabela 9 – Resultados nível 3 com heurística h2_03

6 CONCLUSÃO

Os algoritmos de busca são úteis para diversos tipos de problemas. Para o problema *Elevator logics* as buscas não informadas tem dificuldade em encontrar uma solução, demorando muito tempo para encontrar a solução ou ocasionando erros de memória. Porém a busca informada A* se adequa muito bem ao problema, se a heurística utilizada for acertada.

É possível concluir que o mais importante em problemas de busca é a escolha da heurística, ela pode fazer com que o algoritmo ache a solução em horas ou em segundos. Para fins didáticos e de prática, no problema descrito nesse trabalho qualquer pessoa consegue chegar em uma heurística aceitável pensando no problema por um tempo. No entanto, é preciso ter em mente que em problemas muito específicos, eventualmente é necessário conversar com um profissional da área para que ele indique a melhor forma de escolher um caminho.

Além disso, como os algoritmos foram feitos ”do zero”, foi possível aprender como a estrutura de dados utilizada influencia no tempo de execução do algoritmo, com as funções para checar se um elemento está na estrutura, e ordenar os elementos dentro da estrutura.

Referências

1 NVIDIA. Accelerating AI with GPUs: A New Computing Model. <https://blogs.nvidia.com/blog/2016/01/12/accelerating-ai-artificial-intelligence-gpus/>. Acessado em 11/09/19. Citado na página 1.

2 RUSSEL, P. N. S. **Artificial Intelligence: A Modern Approach. 3rd Edition**. [S.l.: s.n.], 2009. Citado 2 vezes nas páginas 2 e 5.

3 HüBNER, J. - Busca em Espaço de Estados. <http://jomi.das.ufsc.br/ia/busca/busca.pdf>. Acessado em 20/09/19. Citado na página 10.

4 AIMACODE - AIMA python. <https://github.com/aimacode/aima-python>. Acessado em 20/09/19. Citado na página 11.

5 PYTHON - Time complexity. <https://wiki.python.org/moin/TimeComplexity>. Acessado em 13/10/19. Citado 2 vezes nas páginas 11 e 12.

6 STACKOVERFLOW - Time complexity of functions in heapq library. <https://stackoverflow.com/questions/38806202/whats-the-time-complexity-of-functions-in-heapq-library>. Acessado em 13/10/19. Citado na página 12.