

**Unidade Curricular**  
**Inteligência Artificial**  
**2023/2024**

***Problema Big Triple***

**Grupo 9 - P2**

**1. MEMBROS DO GRUPO**

- Davide Pinto 52158
- Nuno Carvalho 72784
- Robim Rodrigues 71385

**2. PROBLEMA E SUA RESOLUÇÃO**

O problema *Big Triple* funda-se em encontrar o triplo de um número utilizando três possíveis operações:

- Incrementar (+1) ao número: Custo 1
- Decrementar (-1) ao número: Custo 2
- Duplicar (x2) o número: Custo 3

Tendo em conta estas operações, pretende-se encontrar o caminho de menor custo entre o número inicial e o número final.

Primeiramente, para a resolução do problema em questão, foi utilizado o algoritmo  $A^*$ , cujo funcionamento consiste em encontrar o caminho mais curto de um nó inicial para um nó final. Durante a sua procura, é utilizada uma função de avaliação  $f^*(n) = g^*(n) + h^*(n)$ , uma combinação do custo acumulado  $g^*(n)$  e uma estimativa do custo desse nó à solução  $h^*(n)$ . Para o problema que foi dado, foi desenvolvida uma função heurística que estimasse o custo de chegar à solução a partir de um dado nó.

De seguida, foi implementado o  $IDA^*$ , um algoritmo variante do  $A^*$  que utiliza uma abordagem de busca em profundidade limitada com heurísticas, onde o limite de profundidade é aumentado gradualmente. O objetivo é encontrar uma solução ótima, como o  $A^*$ , mas com maior eficiência no consumo de memória.

Assim sendo, foram obtidas as seguintes ideias para o algoritmo A\*:

**1) Criar uma árvore de nós ponderado:**

Os Nós representam números.

Arestas permitem operações válidas.

Atribuir a cada aresta um peso que representa o custo da operação.

**2) Definir a heurística apropriada:**

Estimar o custo restante.

Levar em consideração as propriedades do problema.

Heurística implementada especificamente para o problema em questão.

**3) Inicializar estruturas de dados:**

Fila de prioridade para nós a serem explorados.

*Hashmap* para armazenar custo total e nó pai de cada nó visitado.

**4) Definir nó inicial e final:**

Nó inicial é o número fornecido.

Nó final é o triplo do número inicial (*goal*).

**5) Adicionar nó inicial na fila de prioridade:**

Valor de prioridade é igual a  $g^n + h^n$ .

**6) Enquanto a fila de prioridade não estiver vazia:**

Remover nó com maior prioridade.

Verificar se é o nó final.

**7) Se o nó removido for o nó final:**

Recuperar caminho do nó inicial ao nó final.

**8) Caso contrário, expandir o nó removido:**

Gerar os seus sucessores.

Adicionar os sucessores à fila de prioridade.

**9) Atualizar mapa de nós visitados:**

Mover o nó atual para os nós fechados.

**10) Repetir passos 6 a 9 até encontrar o nó final.**

Tendo em conta esta ideologia, foi seguida a sua implementação. Essa implementação da *Best First Search* com A\* para o problema do *Big Triple* é eficiente e garante que o percurso de menor custo seja encontrado.

De seguida, foram imaginadas ideias para a implementação do IDA\* e foram chegados os seguintes passos:

**1) Definir o nó inicial e final:**

Nó inicial é o número fornecido.

Nó final é o triplo do número inicial.

**2) Definir a função heurística:**

Estimar o custo restante para chegar ao nó final.

Utilizar a função de heurística previamente implementada

**3) Definir um limiar inicial:**

Valor do limite de corte inicial é igual a  $h^n$  do nó inicial.

**4) Executar um *loop* até que a solução seja encontrada, ou que tanto a fila de abertos como a *fringe* estejam vazias (caso em que não existe solução):**

**4.1) Executar uma busca em profundidade limitada:**

A partir do nó inicial.

**4.2) Se o nó final for encontrado:**

Retornar o caminho de menor custo.

**4.3) Se o nó final não for encontrado:**

Atualizar o limite para o menor custo encontrado na *fringe*, durante a busca em profundidade limitada.

**5) Repetir o passo 4 até que o nó final seja encontrado ou o limiar seja infinito.**

### 3. UNIT TESTS DESENVOLVIDOS

Os *Unit Tests* foram desenvolvidos no ficheiro *BigTripleUnitTests.java*, que podem ser encontrados dentro da pasta *src*. Foram testados vários números, entre os quais valores grandes, pequenos, pares, ímpares, positivos e negativos.

#### 4. OPÇÕES ESCOLHIDAS DURANTE O DESENVOLVIMENTO

Tendo em conta a ideologia mencionada no ponto 2, seguiu-se à implementação do código.

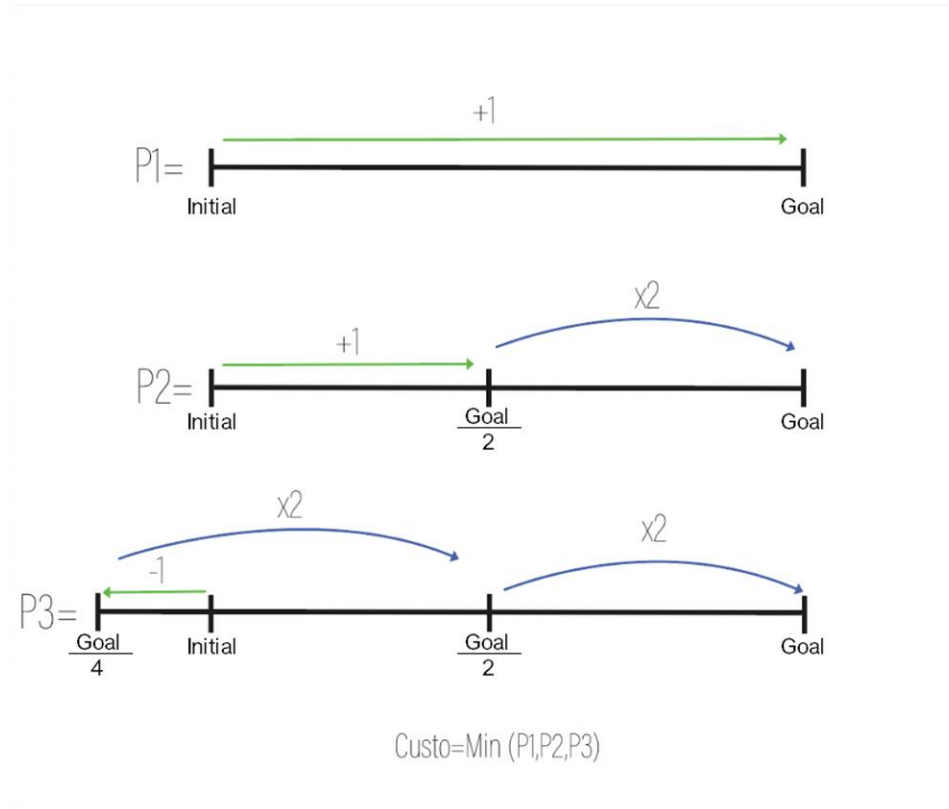


Figura 1 : Esquema da função heurística

Para resolver o problema do *BigTriple*, foi desenvolvida a heurística representada na figura 1 em que para estimar o custo de um nó até ao objetivo são calculados três possíveis custos e retornado o menor. A primeira opção, P1, é o caminho em que se adiciona 1 até chegar ao objetivo. Na segunda opção P2 é adicionado 1, até se chegar a  $\frac{1}{2}$  do objetivo onde é feita uma multiplicação por dois. Por fim, a terceira opção é subtrair 1 até chegar a  $\frac{1}{4}$  do objetivo seguido de duas multiplicações sucessivas por dois. Este processo pode então ser invertido para objetivos menores do que zero.

A\*: implementação idêntica ao *BestFirst*, única diferença sendo a ordenação da fila de abertos com base na função de avaliação  $f^*(n) = g^*(n) + h^*(n)$ .

IDA\*: É determinado um limite inicial igual a  $h^*(n)$  do nó inicial, sempre que são gerados sucessores é testado se estão dentro do limite atual, se estiverem são adicionados à lista de abertos, caso não estejam são adicionados à *fringe*. Quando a lista de abertos está vazia, é verificado se existe algum nó na *fringe*, a *fringe* tal como a lista de abertos é ordenada com base na função de avaliação  $f^*(n) = g^*(n) + h^*(n)$ . É então escolhido o primeiro nó da *fringe* e o valor do limite passa a ser o resultado da função de avaliação

O código fonte inteiro segue a referência do tutorial 1 (*Design-pattern Strategy*) lecionado nas aulas.

## 5. CÓDIGO

É possível encontrar todo o código fonte na pasta *src*.

## 6. JAVADOC

Está inserido no código, em comentários.

## 7. DIAGRAMA DE CLASSES UML

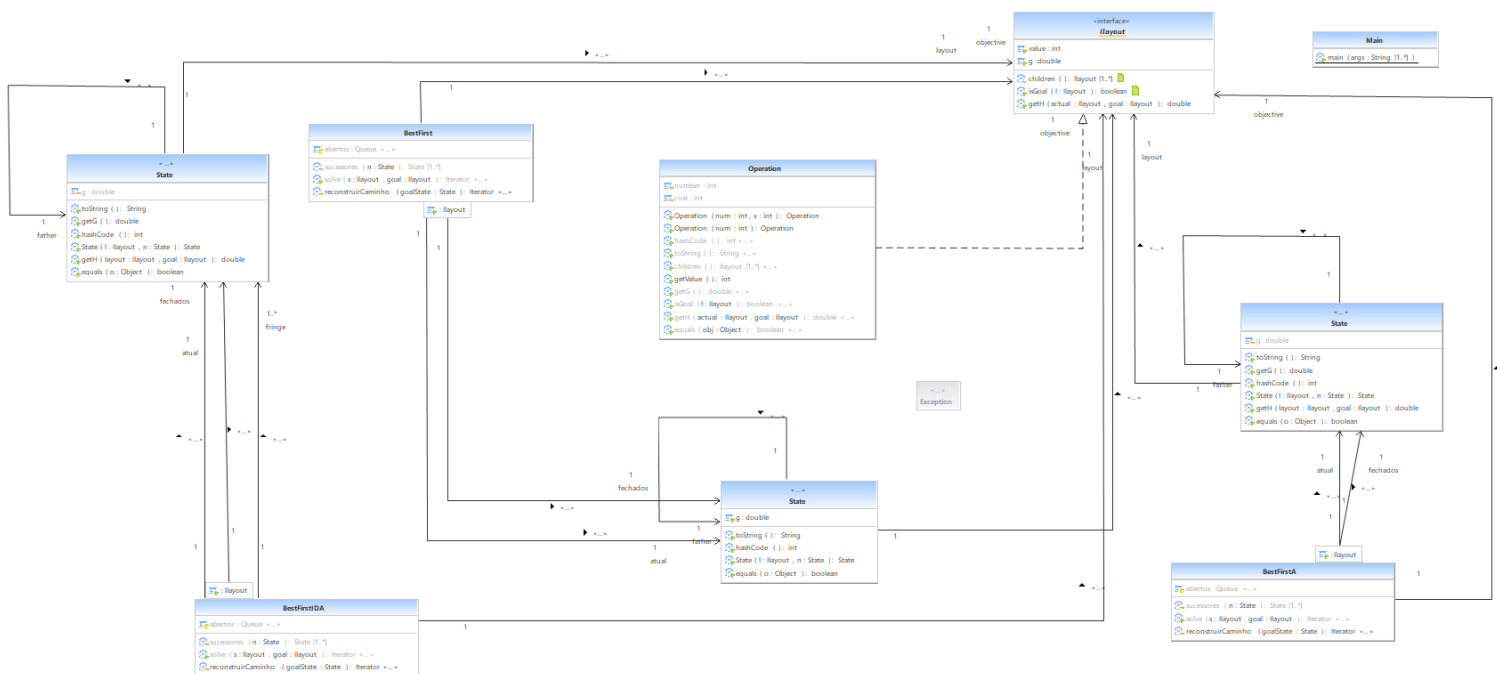


Figura 2 : Diagrama de classes do programa *Big Triple*

## 8. RESULTADOS, ANÁLISES E DISCUSSÕES

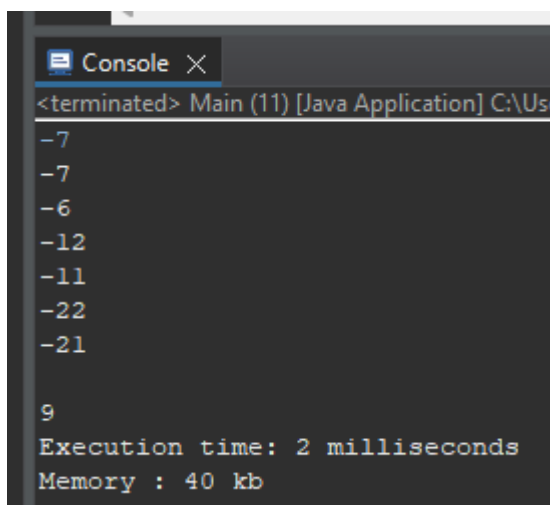
Feito o programa com todas as classes e métodos necessários, obtivemos resultados idênticos aos resultados esperados. Apesar das dificuldades que vieram a aparecer durante a organização e implementação do problema, todos os testes concretizados foram um sucesso. Tendo em conta isso, conseguimos chegar à conclusão de que a utilização do A\* e IDA\* é muito melhor em termos de eficiência, tempo e memória do que o algoritmo *Best-First* utilizado do projeto anterior.

Em termos à utilização do A\* e IDA\*, foi possível observar que algoritmo A\* é mais demoroso do que o IDA\*. Contudo, a memória utilizada para efetuar tais processos é muito menor. A seguinte figura comprova os resultados.

Estado Inicial	Algoritmo	E (Nº Nós expandidos)	G (nº nós gerados)	L(Comprimento da solução)	P = L/E	Solução	tempo execução (ms)	Memória (MB)	Memória (KB)
	C. Uniform	59721	151632	16	0,000267912	16	47	8	8755
54	A*	28	57	28	1	30	2	0	0
	IDA*	28	57	28	1	30	2	0	0
	C. Uniform	Null	Null	Null	Null	Null	Null	Null	Null
-2000	A*	502	1006	502	1	506	3	0	163
	IDA*	502	1006	502	1	506	3	0	122
	C. Uniform	Null	Null	Null	Null	Null	Null	Null	Null
1000	A*	501	1003	501	1	503	4	0	122
	IDA*	501	1003	501	1	503	3	0	122
	C. Uniform	1236	3148	9	0,007281553	9	4	0	491
-23	A*	12	28	9	0,75	13	2	0	0
	IDA*	26	59	9	0,346153846	13	10	0	532
	C. Uniform	Null	Null	Null	Null	Null	Null	Null	Null
12541	A*	6272	12546	6272	1	6274	18	2	2664
	IDA*	12543	25089	6272	0,500039863	6274	25	5	5695
	C. Uniform	Null	Null	Null	Null	Null	Null	Null	Null
231231	A*	115617	231236	115617	1	115618	306	35	36194
	IDA*	231233	462469	115617	0,500002162	115618	118	67	69541
	C. Uniform	Null	Null	Null	Null	Null	Null	Null	Null
50000	A*	25001	50003	25001	1	25003	62	4	4138
	IDA*	25001	50003	25001	1	25003	23	4	4309

Figura 3 : Tabela de resultados utilizando os diferentes algoritmos

Dois Exemplos de *input* e *output*:



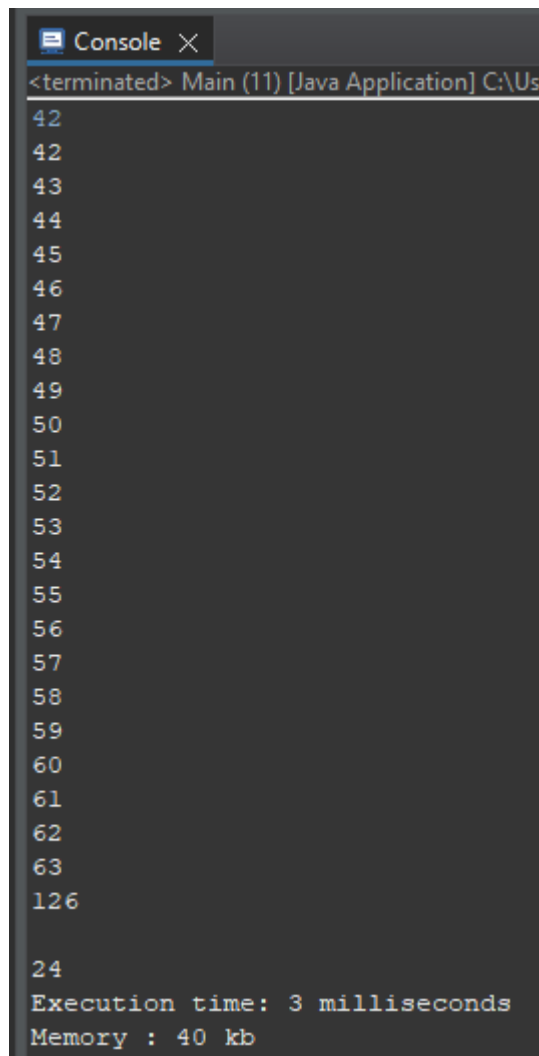
```

Console
<terminated> Main (11) [Java Application] C:\Use
-7
-7
-6
-12
-11
-22
-21

9
Execution time: 2 milliseconds
Memory : 40 kb

```

Figura 4 : Exemplo de *input* e *output* 1



```
Console X
<terminated> Main (11) [Java Application] C:\Us
42
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
126

24
Execution time: 3 milliseconds
Memory : 40 kb
```

Figura 5 : Exemplo de *input* e *output* 2

## 9. CONCLUSÕES OBTIDAS

Tendo observado os resultados, chegamos à conclusão que foi possível alcançar resultados pretendidos com os algoritmos que foram implementados. Além disso, percebemos que ambos os algoritmos, apesar de serem semelhantes, possuem diferenças importantes que fazem com que ambos os algoritmos se diferenciem um do outro em termos de eficiência, tempo e memória. Portanto, para ambos os algoritmos A\* e IDA\*, foi então possível chegar a uma lista com as suas principais diferenças.

- **Para o tipo de algoritmo:**
  - Enquanto que o A\* é um algoritmo de busca informada (ou heurística) que utiliza uma função de avaliação para encontrar o caminho mais curto de um nó inicial para um nó de destino, o IDA\* é uma variante desse algoritmo. Utiliza uma combinação com a busca em profundidade iterativa com a estratégia heurística do A\*, explorando gradualmente caminhos mais longos em cada iteração.

- **Complexidade de memória:**

- A complexidade de memória do A\* pode ser alta, pois esse algoritmo mantém uma lista de nós a serem explorados, o que é capaz de ocupar muito espaço em problemas com grandes espaços de estados. Para o IDA\*, o algoritmo é mais eficiente em termos de memória, pois realiza a busca em profundidade com um limite de profundidade em cada iteração, evitando a necessidade de armazenar uma lista completa de nós. Para o caso utilizado na resolução deste problema, com a heurística implementada pelo grupo, não existe benefício em utilizar o algoritmo IDA\*.

- **Eficiência em Grandes Espaços de Estados**

- A\* pode ser menos eficiente em problemas com grandes espaços de estados devido à sua alta complexidade de memória e à sua necessidade de armazenar uma grande quantidade de nós na lista de exploração. Por sua vez, o IDA\* é mais adequado para problemas com grandes espaços de estados, pois controla a complexidade de memória e explora caminhos mais longos de forma iterativa, tornando-o mais eficiente em termos de memória. Para o problema em questão, não existe qualquer benefício devido à heurística utilizada no problema.

Dado estas diferenças, se fosse apresentada a questão de qual dos algoritmos seria melhor, não haveria uma resposta definitiva, pois a escolha entre A\* e IDA\* depende do problema apresentado e dos seus requisitos e restrições. Tal como mencionado acima, A\* consome mais memória, mas pode ser mais rápido para problemas menores, enquanto IDA\* é eficiente em termos de memória e adequado para problemas maiores devido à sua abordagem iterativa de limitação de profundidade.

A nosso ver, a principal dificuldade foi encontrar uma heurística adequada para resolver o problema. Podemos mencionar que houve aprendizagem nos tópicos de Inteligência Artificial, mais especificamente no funcionamento dos algoritmos que foram utilizados.

Apesar de tudo, com este trabalho progredimos bastante na nossa aprendizagem e o problema que acabamos de enfrentar permitiu-nos perceber mais sobre o funcionamento da cadeira e a compreensão do mundo da programação no geral.

## 10. REFERÊNCIAS BIBLIOGRÁFICAS

- <https://iq.opengenus.org/a-search/>
- <https://www.geeksforgeeks.org/iterative-deepening-a-algorithm-ida-artificial-intelligence/>
- <https://ai.stackexchange.com/questions/24682/when-does-ida-consider-the-goal-has-been-found>
- [https://en.wikipedia.org/wiki/Iterative\\_deepening\\_A\\*](https://en.wikipedia.org/wiki/Iterative_deepening_A*)



