RELATÓRIO FINAL - INTELIGÊNCIA ARTIFICIAL
DCC014 - 2020.1 - ERE

Grupo 2: **Davi Rezende Domingues**
Pedro Cotta Badaró
Jonas Gabriel Maia

Professor: **Saulo Moraes**

Tema: ***Jogo dos Jarros***

1. Descrição do Problema

O Jogo dos Jarros trata de um problema onde temos uma quantidade n de jarros, e cada jarro possui uma capacidade c. O problema consiste em manipular quantidades de água entre os jarros, a fim de se alcançar um estado objetivo.

A manipulação de água entre os jarros ocorre através dos operadores “Encher Jarro”, “Esvaziar Jarro” e “Transferir Água”. Um jarro só pode ser enchido caso não esteja com sua capacidade máxima ocupada. Um jarro só pode ser esvaziado caso contenha alguma quantidade de água. Uma transferência só pode ocorrer caso exista alguma quantidade de água no jarro de origem e caso o jarro de destino não esteja cheio. Ainda sobre o operador de transferência, temos que essa operação pode ocorrer de diversas maneiras, entre diversos jarros. Portanto, deve ser definido quais operações de transferência serão válidas, em cada implementação do problema.

O estado objetivo é pré-definido pelo implementador ou pelo usuário, podendo ser atingir a metade da capacidade de algum jarro ou atingir uma determinada quantidade de água em qualquer um dos jarros, por exemplo.

Todos os jarros estão, inicialmente, vazios, e, através dos operadores, são gerados novos estados, até que um estado objetivo seja identificado.

2. Descrição da Implementação

2.1. Detalhes da Implementação

Nosso código foi desenvolvido separado em arquivos. Temos um arquivo para cada algoritmo implementado, além de um arquivo Graph.py para as 3 classes que compõem o grafo, um arquivo Jug.py para a classe que representa os jarros, e um arquivo Main.py, onde temos nossa função principal, que contém a leitura do arquivo de entrada, a interface com o usuário e a chamada para todos os algoritmos implementados.

Para o jogo dos jarros, nós definimos a seguinte estratégia de controle em nossa implementação:

R1: Encher Jarro.
R2: Transferir para a esquerda.
R3: Transferir para a direita.
R4: Esvaziar jarro.

Para cada algoritmo foram analisados o tempo de execução, o número de nós gerados e expandidos, a profundidade do grafo e, naqueles que utilizam esta métrica, a heurística e o custo da solução obtida. Além disso, a execução de cada algoritmo imprime na tela o caminho percorrido passo a passo pelos operadores, mostrando toda a extensão do grafo de estados gerado.

2.2. Ferramentas Utilizadas

Nosso trabalho foi desenvolvido em Python, sendo utilizado o VSCode. Para as reuniões, foi utilizada a plataforma Google Meet, e o controle de versões do código foi feito através do GitHub.
Tanto o relatório final, quanto os slides da apresentação, foram desenvolvidos através do Google Docs.

2.3. Formato de Entrada

A entrada dos dados é feita através de um arquivo de entrada em formato .txt. A primeira linha indica a quantidade de água objetivo em qualquer jarro, enquanto que as linhas seguintes indicam as capacidades de cada jarro.Isso permite que nossa configuração de execução seja completamente dinâmica. A **Figura 1** mostra um exemplo de arquivo de entrada, onde temos 2 jarros, um com capacidade 4, outro com capacidade 3, e o objetivo é atingir um volume igual à 2 em qualquer um dos jarros.

Figura 1: exemplo de arquivo de entrada, em formato .txt

2.4. Estruturas e Classes

Nosso código trabalha com uma estrutura de grafos, que envolveu a criação de 4 classes: node, edge, graph e jug. Vale ressaltar que indicamos aqui os inicializadores como construtores para melhor entendimento.

A classe 'node' representa os nós do grafo de estados, e contém um vetor de jarros (objetos do tipo 'Jug'), um nó pai e um atributo que mede a profundidade do nó, além de peso e heurística (utilizado em algoritmos de busca que trabalham com estes atributos). A **Figura 2** mostra a estrutura dessa classe:

```
class Node:
    #construtor
    def __init__(self, parent_node, jug_arr):
        self._parent_node = parent_node
        self._jug_arr = jug_arr
        self._heuristic=0
        self._weight=0
        self._depth=0
```

Figura 2: estrutura da classe 'Node'

A classe 'Edge' representa as arestas do grafo de estados, e contém o nó de origem, o nó destino e o operador aplicado. A **Figura 3** mostra a estrutura dessa classe:

```
class Edge:
    #construtor
    def __init__(self, origin, destiny, generating_rule):
        self._origin = origin
        self._destiny = destiny
        self._generating_rule = generating_rule
```

Figura 3: estrutura da classe 'Edge'

A classe ‘Graph’ representa o próprio grafo, e contém o nó raiz (root), uma lista de nós e uma lista de arestas. A **Figura 4** mostra a estrutura dessa classe:

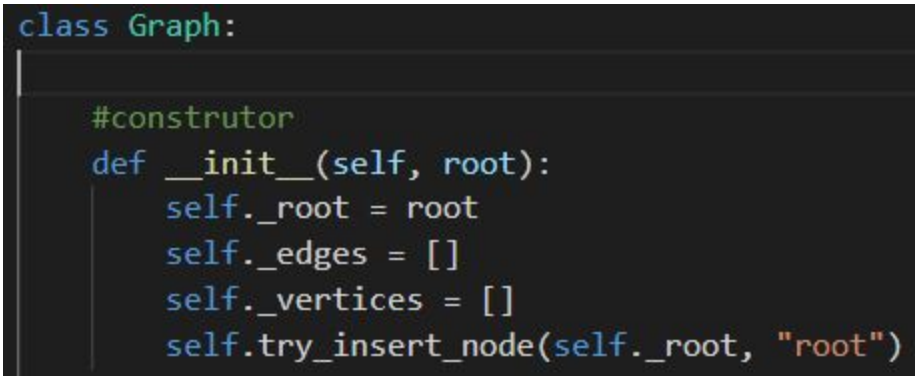


Figura 4: Estrutura da classe ‘Graph’

A classe ‘Jug’ representa os jarros, sob os quais os operadores são aplicados. Possui atributos que indicam a capacidade do jarro e a quantidade de água atual do jarro. A **Figura 5** mostra a estrutura dessa classe:

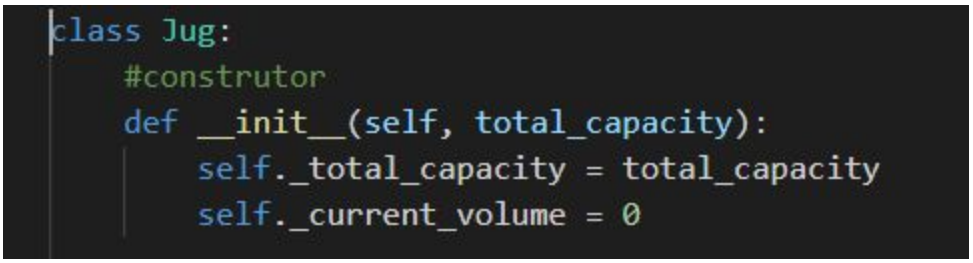


Figura 5: Estrutura da classe ‘Jug’

2.5. Funções, Operações e Métodos

Como visto na seção anterior, nosso programa é estruturado em cima de 4 classes principais. Nesta seção será comentado mais a fundo sobre os métodos dessas classes, além de outras funções e métodos presentes no código.

A classe Graph possui a seguinte lista de métodos:

- **_init_(root)** = Construtor da classe .
- **get_vertices()** = retorna o número de vértices.
- **get_edges()** = retorna o número de arestas.
- **print_graph()** =Imprime as arestas do Grafo.
- **try_insert_node(node,generating_rule)** = verifica se o nó está no grafo, insere se não estiver.
- **insert_node_LP(node,generating_rule)** = usado para inserir nós repetidos.
- **check_insert_node(node)** = verifica se o nó está no grafo.

A classe Edge possui a seguinte lista de métodos:

- **_init_(node,generating_rule)** = Construtor da classe Edge.
- **get_origin()** = retorna o nó de origem.
- **Get_generating_rule()** = retorna o nó de destino.
- **get_destiny()** = retorna o nó de destino.
- **printEdge()** = Imprime as arestas do nó.

A classe Node possui a seguinte lista de métodos:

- **_init_(parent_node,jug_arr)** = Construtor da classe Node.
- **get_heuristic()** = retorna valor da heurística.
- **get_weight()** = retorna o peso do nó.
- **set_weight()** = altera o peso do nó.
- **get_parent_node()** = retorna nó pai.
- **get_jug_arr()** = retorna vetor de jarros.
- **set_jug_arr()** = altera vetor de jarros.
- **get_node_state()** = retorna valor do estado do nó.
- **transfer_to_right(jug_pos)** = recebe a posição do jarro e chama função transfer_to passando como parâmetros sua posição e a posição do jarro da direita.
- **transfer_to_left(jug_pos)** = recebe a posição do jarro e chama função transfer_to passando como parâmetros sua posição e a posição do jarro da esquerda.
- **transfer_to(source_jug_index,target_jug_index)** = transfere água do jarro inicial para o jarro destino.
- **control_strategy(operator,index)** = Recebe qual regra e em qual índice do jarro ela será utilizada. Ela então executa a regra escolhida.
- **is_solution(target)** = retorna se a solução foi encontrada ou não.
- **try_apply_rule(g)** = recebe um grafo, executa a função control_strategy e retorna qual regra foi utilizada no nó.

A classe Jug possui a seguinte lista de métodos:

- **_init_(total_capacity)** = Construtor da classe Jug.
- **get_total_capacity()** = retorna a capacidade total
- **Get_current_volume()** = retorna o volume atual.
- **set_current_volume(value)** = altera o volume atual.
- **spill()** = esvazia o jarro.
- **fill()** = enche o jarro.

Cada algoritmo implementado foi desenvolvido em um arquivo Python à parte, como uma função. A seguir estão os detalhes de cada uma:

Arquivo backtracking.py:

- **backtracking(s, target) =** função que recebe como parâmetro um nó inicial (raiz, com os jarros vazios) e o objetivo, e implementa uma busca em forma de backtracking, imprimindo resultados e execução na tela.

Arquivo largura.py:

- **largura(s, target) =** função que recebe como parâmetro um nó inicial (raiz, com os jarros vazios) e o objetivo, e implementa uma busca em largura, imprimindo resultados e execução na tela.

Arquivo profundidade.py:

- **profundidade(s, target) =** função que recebe como parâmetro um nó inicial (raiz, com os jarros vazios) e o objetivo, e implementa uma busca em profundidade, imprimindo resultados e execução na tela.

Arquivo ordenada.py:

- **ordenada(s, target) =** função que recebe como parâmetro um nó inicial (raiz, com os jarros vazios) e o objetivo, e implementa uma busca ordenada, imprimindo resultados e execução na tela.

Arquivo guloso.py:

- **guloso(s, target) =** função que recebe como parâmetro um nó inicial (raiz, com os jarros vazios) e o objetivo, e implementa uma busca gulosa, imprimindo resultados e execução na tela.

3. Resultados e Execuções

A seguir vamos mostrar os resultados que obtemos para uma série de execuções testes que realizamos. Para cada algoritmo foram analisados o tempo de execução, o número de nós gerados e expandidos, a profundidade do grafo e, naqueles que utilizam estas métricas, a heurística e o custo da solução obtida. As instâncias estão representadas no formato “(a,b,...w) com objetivo z”, onde a,b...w são as capacidades de cada jarro, e z é a quantidade objetivo de se alcançar, em qualquer jarro.

Instância: (4,3) com objetivo 2 -> SUCESSO, POSSUI SOLUÇÃO

ALGORITMO	Tempo de execução (s)	Nós gerados	Nós expandidos	Profundidade	Custo
Backtracking	0.021	7	7	7	X
Largura	0.018	10	9	5	X
Profundidade	0.008	7	7	7	X
Ordenada	0.119	18	14	4	24
Guloso	0.011	10	8	6	X

Instância: (4,3,5) com objetivo 7 -> FRACASSO, NÃO POSSUI SOLUÇÃO

ALGORITMO	Tempo de execução (s)	Nós gerados	Nós expandidos	Profundidade	Custo
Backtracking	8.660	95	95	47	X
Profundidade	0.144	56	55	55	X
Largura	0.747	96	95	7	X
Guloso	0.842	96	95	9	X
Ordenada	18.184	473	472	10	X

Instância: (4,3,5) com objetivo 2 -> SUCESSO, POSSUI SOLUÇÃO

ALGORITMO	Tempo de execução (s)	Nós gerados	Nós expandidos	Profundidade	Custo
Backtracking	0.074	10	10	10	X
Profundidade	0.023	10	10	10	X
Largura	0.044	25	11	3	X
Guloso	0.078	39	18	6	X
Ordenada	0.062	33	11	3	33

Instância: (4,3,5,6) com objetivo 2 -> SUCESSO, POSSUI SOLUÇÃO

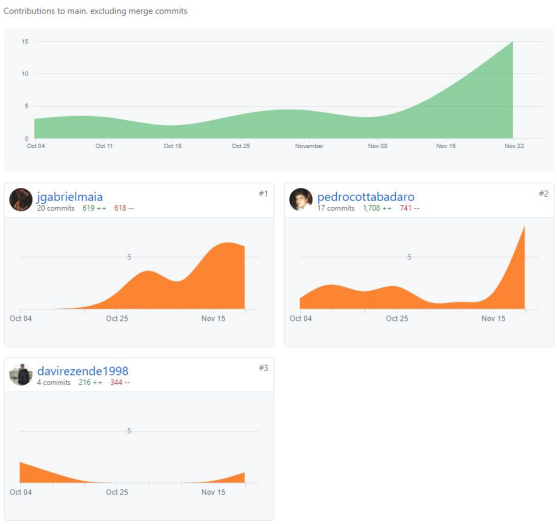
ALGORITMO	Tempo de execução (s)	Nós gerados	Nós expandidos	Profundidade	Custo
Backtracking	0.06	4	4	4	X
Largura	0.170	44	14	3	X
Profundidade	0.0009	4	4	4	X
Ordenada	0.188	56	12	3	33
Guloso	0.369	90	28	7	X

Instância: (4,3,5,6) com objetivo 7 -> FRACASSO, NÃO POSSUI SOLUÇÃO

ALGORITMO	Tempo de execução (s)	Nós gerados	Nós expandidos	Profundidade	Custo
Backtracking	Excedeu o limite de recursões	Excedeu o limite de recursões	Excedeu o limite de recursões	Excedeu o limite de recursões	X
Largura	41.883	720	719	11	X
Profundidade	5.752	337	336	336	X
Ordenada	Excedeu o limite de recursões	Excedeu o limite de recursões	Excedeu o limite de recursões	Excedeu o limite de recursões	Excedeu o limite de recursões
Guloso	69.927	720	719	16	X

4. Divisão de Tarefas

Nosso grupo trabalhou sempre de forma 100% conjunta, em reuniões semanais. Todo o código foi desenvolvido através de encontros virtuais, onde os membros opinaram e contribuíram de forma colaborativa com o projeto. Embora os commits possam estar concentrados em um membro do grupo, ou outro, o trabalho foi desenvolvido 100% de forma colaborativa. O Google Meet e o GitHub foram amplamente utilizados.
O mesmo se deu para a elaboração dos slides de apresentação e deste relatório, um trabalho colaborativo através de documentos compartilhados no Google Docs.
O gráfico a seguir é referente ao código produzido, mas não reflete de maneira alguma a participação individual dos membros. Todas as funcionalidades foram desenvolvidas em conjunto, seja em dupla ou trio.



5. Dificuldades Encontradas

Inicialmente, nosso grupo enfrentou uma grande dificuldade em entender como trabalhar com o problema no código. Não estava claro se trabalharíamos com uma orientação a objetos, tratando cada estado ou jarro como instâncias isoladas, e a definição de uma forma de implementação foi bastante custosa. Uma vez definida a estrutura inicial da nossa implementação, desenvolver os algoritmos dos métodos foi uma tarefa relativamente tranquila, baseando sempre nos pseudocódigos dos materiais fornecidos pelo professor.

A maior dificuldade após a definição da estrutura inicial do código foi implementar o primeiro algoritmo, o backtracking. Não por conta da complexidade do algoritmo, mas porque foi o primeiro a ser implementado, portanto cada passo foi um desafio. Ao final, foram implementados os métodos necessários, o que facilitou a implementação dos outros algoritmos pedidos pelo professor.

Outra dificuldade encontrada foi trabalhar com as noções de peso e heurística em um problema de jogo de jarros, visto que não vimos nenhum exemplo como esse em sala de aula, e arestas com peso nos nossos grafos de estado não são tão intuitivas como em outros grafos, como o mapa de uma cidade, por exemplo.

O backtracking e o algoritmo de busca ordenada apresentaram falha ao tentar encontrar soluções inviáveis (fracasso esperado). Nossa pretensão era criar algoritmos robustos tanto em casos de soluções viáveis como em casos onde não há solução, porém se faz necessário ainda realizar testes minuciosos para resolver questões relacionadas à recursividade em nossos algoritmos.

6. Conclusões

Concluímos que os algoritmos ensinados pelo professor Saulo Moraes são bastante eficientes para encontrar soluções nas instâncias do problema que testamos, cada um com suas características.

Por conta do tempo apertado nesse período de Ensino Remoto Emergencial, nosso grupo não conseguiu desenvolver a tempo os algoritmos de Busca A* e Busca IDA*. Se tivéssemos mais tempo acreditamos que conseguiríamos sem maiores dificuldades, o que enriqueceria nossos experimentos e resultados obtidos.