

PUCRS – Escola Politécnica  
Ciência de Dados e Inteligência Artificial – 2022/2

# Relatório T1 – Algoritmos e Estruturas de Dados II

Thiago de Almeida Macedo – 21104690

Porto Alegre – RS

## Introdução

---

Este relatório tem como objetivo descrever o trabalho 1 da disciplina de Algoritmos e Estruturas de Dados II, explicando o problema que foi abordado e a solução elaborada, mostrando detalhes de sua implementação em Python e demonstrando sua eficácia através de casos de testes.

## Enunciado do Trabalho

---

O trabalho consistia em encontrar a quantidade mínima de movimentos necessários para distribuir uma certa quantia de água entre três jarros distintos. Cada um deles possuía três valores:

- Capacidade de água (no máximo 40 litros)
- Quantidade de água inicial
- Quantidade de água desejada

O objetivo era, a partir da quantidade inicial de cada jarro, passar água de um para o outro até que todos os três estivessem com a quantidade desejada, fazendo o menor número de movimentos (troca de água entre jarros) possível, sem quebrar nenhuma das seguintes regras:

- É proibido jogar água fora.
- É proibido pegar água de uma fonte
- Só é possível esvaziar um jarro em outro ou completar o outro até a borda

O programa desenvolvido deveria ler de um arquivo os dados necessários para a criação de três jarros e calcular a menor número de movimentos necessários para se chegar na solução, escrevendo-os em um arquivo de saída.

## A Solução Desenvolvida

---

Para resolver o problema proposto, desenvolvi um algoritmo que para uma dada configuração de jarros, constrói uma árvore explorando todos os movimentos possíveis até encontrar a solução ou avisa que não há nenhuma, quando o problema for impossível. Seu funcionamento ocorre da seguinte forma:

1. Pega uma configuração (três jarros com seus respectivos valores)
2. Para essa configuração, cria um novo nodo para cada troca de água possível. (esse passo gera no máximo 6 novos nodos)
3. Para cada nodo criado no passo anterior, verifica se ele é a solução:
  - a. Se for: retorna o nível do nodo
  - b. Se não for: vai para o próximo passo
4. Novamente, cria um nodo diferente para cada uma das trocas de água possível para cada um dos nodos e volta para o passo 3.

Seguindo esse algoritmo, eventualmente será criado um nodo que contém a água distribuída entre os jarros da maneira correta. Quando esse nodo for criado, o algoritmo para e retorna seu nível, representando a quantidade de movimentos necessários para alcançá-lo.

Embora seja garantido o encontro de uma solução se ela existir, o algoritmo ainda sim apresenta problemas. Se, a cada novo nível, criarmos um novo jarro para cada um dos movimentos de água possível, o número de novos jarros será igual a quantidade no nível anterior multiplicada por 6. Isso gera problemas à medida que a quantidade de movimentos para encontrar a solução cresce.

Para evitar que o programa crie uma árvore grande demais, foi necessário criar uma lista de estados, que servia para armazenar os estados de distribuição de água que já haviam sido explorados e dessa forma não criar nodos repetidos. Assim, a quantidade de nodos criados não crescia tão de pressa, possibilitando a árvore de explorar mais distribuições.

Para desenvolver esse algoritmo e achar a solução foi necessário o desenvolvimento de um extenso programa em Python, com diversas classes e métodos que em conjunto, gerassem a árvore de maneira correta.

## Principais Métodos da Solução

---


1. **Water Dump:** Despeja água de um jarro para outro, completando o que recebe ou esvaziando o que entrega.

```
1  @staticmethod
2  def water_dump(jar_gives, jar_receives):
3      """Puts water from a jar to another
4
5      Args:
6          jar_gives (Jar): jar that will give the water
7          jar_receives (Jar): jar that will receive the water
8
9      Returns:
10         Jar: both jars with new amounts of water
11     """
12     jar_receives_remaining_capacity = jar_receives.capacity - jar_receives.current_amount
13
14     if (jar_gives.current_amount == jar_receives_remaining_capacity) or (jar_gives.current_amount < jar_receives_remaining_capacity):
15         jar_receives.add_water(jar_gives.current_amount)
16         jar_gives.remove_water(jar_gives.current_amount)
17     else:
18         jar_receives.add_water(jar_receives_remaining_capacity)
19         jar_gives.remove_water(jar_receives_remaining_capacity)
20
21     return jar_gives, jar_receives
```

2. **Build Tree For:** Recebe um nodo e constrói uma árvore de sucessão com um novo nodo para cada despejo de água possível a partir da configuração do inicial. A árvore criada tem fica com o nodo original na raiz e todos os despejos como filhos. Para qualquer distribuição de água inicial, o número de filhos sempre fica entre zero, quando os estados dos filhos já existem na árvore ou quando não é possível fazer nenhum despejo e seis, quando o contrário acontece.

```
1 def _build_tree_for(self, n: Node) -> None:
2     """Builds the sucession tree for a given node
3
4     Args:
5         n (Node): node that we wan't the sucession tree for
6     """
7     i = 0
8     for node in n.create_copies():
9         combination = node.combinations()[i]
10        if node.check_water_dump_possibility(combination[0], combination[1]):
11            node.water_dump(combination[0], combination[1])
12            if node.state() not in self.states:
13                self.add(node, parent=n)
14            else:
15                del node
16        else:
17            del node
18        i+=1
19
```

3. **Build Solutions Tree:** Para uma dada lista de nodos, checa se cada um representa a solução. Se algum deles for, devolve o nível do nodo. Caso o contrário, traz o método **'Build Tree For'** para cada um deles, adicionando os nodos gerados em uma nova lista e chama a si mesmo para essa nova lista.




```

1  def _build_solutions_tree(self, node_list: list):
2      """Builds the solution tree for a list of nodes
3
4      Args:
5          node_list (list): list of nodes that the tree will be built for
6
7      Returns:
8          bool: if it has reached a solution node
9      """
10     next_nodes = []
11     for node in node_list:
12         if check_solution(node) == True:
13             return self.level(node)
14     for node in node_list:
15         self._build_tree_for(node)
16         for child in node.children:
17             next_nodes.append(child)
18     return self._build_solutions_tree(next_nodes)

```

4. **State:** Para um determinado nodo, devolve o estado (quantidade de água em cada jarro) em formato de tupla: (quantidade 1, quantidade 2, quantidade 3).




```

1  def state(self):
2      """Current amount of water in each jar for a node
3
4      Returns:
5          tuple: tuple with the 3 amount of water for the three nodes
6      """
7      return tuple([self.jar1.current_amount, self.jar2.current_amount, self.jar3.current_amount])

```

5. **Check Solution:** Para um determinado nodo, verifica se ele é a solução do problema, ou seja, para cada jarro, a quantidade de água atual é a mesma da quantidade de água desejada.




```

1  def check_solution(node: Node):
2      """Function to check if the node has reached the solution, that is, for all nodes,
3          the current amount of water is equal to the desired amount of water
4
5      Args:
6          node (Node): node that will be tested
7
8      Returns:
9          bool: if the node is the solution
10     """
11     jar1_done = node.jar1.current_amount == node.jar1.desired_amount
12     jar2_done = node.jar2.current_amount == node.jar2.desired_amount
13     jar3_done = node.jar3.current_amount == node.jar3.desired_amount
14
15     return jar1_done and jar2_done and jar3_done

```

6. **Build Tree:** Método principal do programa, a partir da raiz da árvore, constrói toda a árvore de busca e devolve o nível do nodo que contém a solução. Basta chamá-lo para obter a resposta.



```

1  def build_tree(self):
2      """Finds the solution for a tree
3
4      Returns:
5          int: Amount of movements necessary to reach the solution
6      """
7      return self._build_solutions_tree([self.root])

```

## Resultados dos Testes

Para a avaliação do programa, era necessário ler um arquivo que continha uma série de configurações de jarros e escrever, em um novo arquivo, a configuração e sua devida resposta. O arquivo de entrada se chamava: *‘entrada\_exemplo\_T1.txt’* e o de saída foi nomeado *‘saída\_T1.txt’*. Os dois arquivos são exatamente iguais, exceto pelo fato de o arquivo de saída possuir o menor número de movimentos possíveis para se encontrar a solução, da seguinte forma:



## Conclusão

O trabalho 1 da disciplina de Algoritmos e Estruturas de Dados II foi extremamente desafiador e exigente, sendo necessárias muitas horas de trabalho. A solução foi completamente desenvolvida em Python 3 através do VS Code. A solução também está disponível em um repositório do GitHub [neste link](#).