Primeira solução

Após a discussão da dupla sobre a implementação do problema, decidiu-se trabalhar com a linguagem Python. Para o problema foram feitos 3 módulos: “reader.py”, o qual contém a função “get\_cases” para leitura dos arquivos de teste no formato de texto; “Graph.py” o qual contém a classe “Graph”, cujo será a estrutura implementada pela dupla para resolver o problema; Além de “main.py”, onde será executada a solução em sua ordem correta.

Como já mencionado a dupla escolheu a estrutura de grafo para resolver o problema dos jarros, onde iremos nos referir ao conjunto dos nodos em uma determinada altura como estado, além dos nodos sendo a quantidade de agua presente nos 3 jarros naquele determinado nodo. Essa estrutura é inicializada com um vetor contendo o estado inicial, outro vetor sendo o estado final esperado e um vetor sendo o tamanho de cada jarro. Os atributos dessa estrutura, além dos já mencionados, são: Um vetor “actual\_state” que seria o conjunto dos estados que estamos observando em determinada altura do grafo; Um vetor “closed\_states”, contendo todos os nodos já antes observados; Um vetor “last\_states” sendo o conjunto dos estados na ultima altura analisada até o momento; Um boleano “token” para que o algoritmo saiba a hora de parar de procurar uma solução.

Graph{

initial\_state <- estados iniciais dos jarros

final\_state <- estados finais dos jarros

jars\_len <- tamanho dos jarros

actual\_state <- estados iniciais dos jarros

closed\_states <- lista vazia

last\_states <- estados iniciais dos jarros

token <- Falso

}

Em seguida, chamamos o procedimento “get\_next\_states”, onde a partir de cada nodo presente no vetor último estado, serão gerados os próximos estados a partir de todas as possíveis permutações dos jarros aplicados as funções de transferência de água “fill\_all\_jar” e “drain\_jar\_out”. Quando algum nodo retornar de alguma dessas funções igual ao nodo esperado no estado final, o algoritmo para e mostra a quantidade de movimentos que foi necessário para executa-lo. Caso não encontre duas coisas poderão acontecer. A primeira é, caso as funções não consigam criar nenhum nodo não previamente existente, o algoritmo termina e é dado como impossível. A segunda é, ele retornará nodos não previamente existentes e isso gerará o estado atual do grafo, que por consequência será o próximo estado passado. Como bem podemos ver no pseudo-código:

procedimento get\_next\_states

i <- 1

Enquanto token = falso:

para cada state em last\_states faça

combinations <- permutations([0, 1, 2], 2)

closed\_states.insert(state)

para cada combination em combinations faça

se fill\_all\_jar(state, combination) = true entao

token <- True

pare

fim

se drain\_jar\_out(state, combination) = true entao

token <- True

pare

fim

fim

se actual\_state = [] entao

print("Impossivel")

pare

fim

last\_states <- actual\_state

actual\_state <- []

se token = True entao

print("Terminou com ", i ,"movimentos")

fim

i <- i + 1

fim

fim

fim

A vantagem dessa função nesse formato é a impossibilidade de formação de novos nodos se eles já existirem em algum estado, o que faz com que o grafo não cresça e consiga encontrar soluções impossíveis.

A função anteriormente descrita usufrui das funções de transferência de conteúdo de jarros “fill\_all\_jar” e “drain\_jar\_out”. Ambas funções cobrem as especificações de transferência de conteúdo dadas no problema, onde “fill\_all\_jar” a partir de um jarro enchemos outro jarro até o seu máximo, e a função “drain\_jar\_out” esvaziamos todo um jarro em outro. Lembrando de em nenhuma dessas funções exceder o máximo de conteúdo que um jarro aguenta, conforme seus pseudo-códigos:

Procedimento drain\_jar\_out(state:list, indexes:tuple)

se (state[indexes[0]] + state[indexes[1]] <= jars\_len[indexes[0]]) e (não(state\_[indexes[1]]=0))) faça

state[indexes[0]] <- state[indexes[0]] + state[indexes[1]]

state[indexes[1]] <- 0

se state = final\_state faca

retorne True

fim

se state não em closed\_states faca

actual\_state.insert(state)

fim

fim

fim

procedimento fill\_all\_jar(state:lista, indexes:tupla)

se jars\_len[indexes[0]] - state[indexes[0]] < state[indexes[1]] entao

missing <- jars\_len[indexes[0]] - state[indexes[0]]

state[indexes[0]] <- state[indexes[0]] + missing

state[indexes[1]] <- state[indexes[1]] - missing

se state = final\_state entao

retorne True

fim

se state não em closed\_states entao

actual\_state.insert(state)

fim

fim

fim

Resultado

Iremos comparar os resultados com a quantidade de movimentos e tempo de execução do programa na linguagem Python:

Terminou com 1 --- 0.0 seconds --- Terminou com 4 --- 0.0 seconds --- Terminou com 7 --- 0.015623331069946289 seconds --- Terminou com 10 --- 0.0 seconds --- Terminou com 12 --- 0.0 seconds --- Terminou com 15 --- 0.0 seconds --- Terminou com 18 --- 0.015622615814208984 seconds --- Terminou com 20 --- 0.0 seconds --- Terminou com 22 --- 0.015624284744262695 seconds --- Terminou com 28 --- 0.0 seconds --- Impossivel --- 0.0 seconds ---