UNIVERSIDADE DE ÉVORA

Relatório - 1º Trabalho - Estruturas de Dados e Algoritmos II

Trabalho realizado pelo grupo g123 (Mooshak) composto por:

- Rui Roque nº42720
- Tomás Dias nº42784

Introdução ao problema

Este trabalho tem como objetivo implementar uma solução para o problema *Cod Fishing*.

Para tal, foi pedido um programa que dado as localizações (coordenadas) e ratings dos barcos bem como as localizações e quantidade de peixe existente dos portos, procurava obter o total de peixe capturado, a distância percorrida pelos barcos até aos portos e o total de ratings dos barcos que foram atribuídos a um porto.

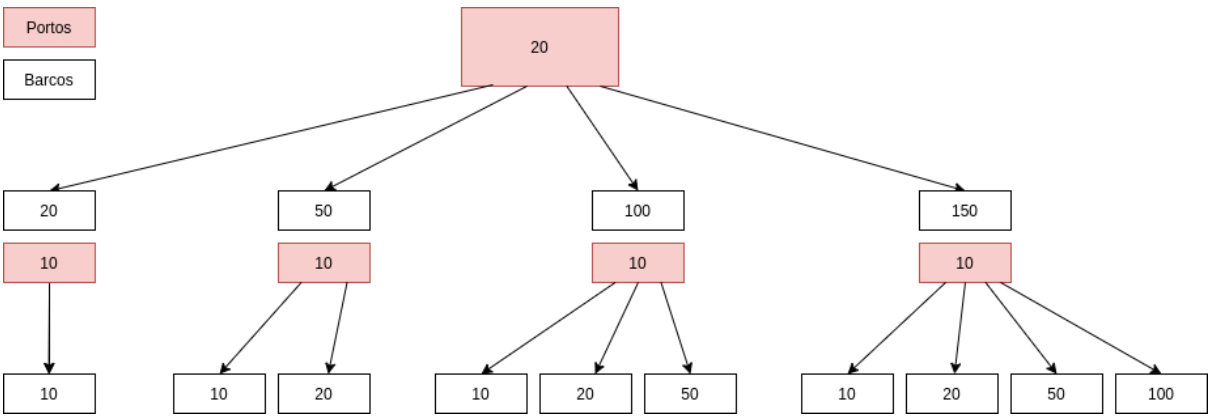
O programa teria de funcionar de forma a que primeiro se maximize o total de peixe capturado, segundo se minimize a soma das distâncias percorridas pelos barcos e por fim se minimize a soma dos ratings dos barcos.

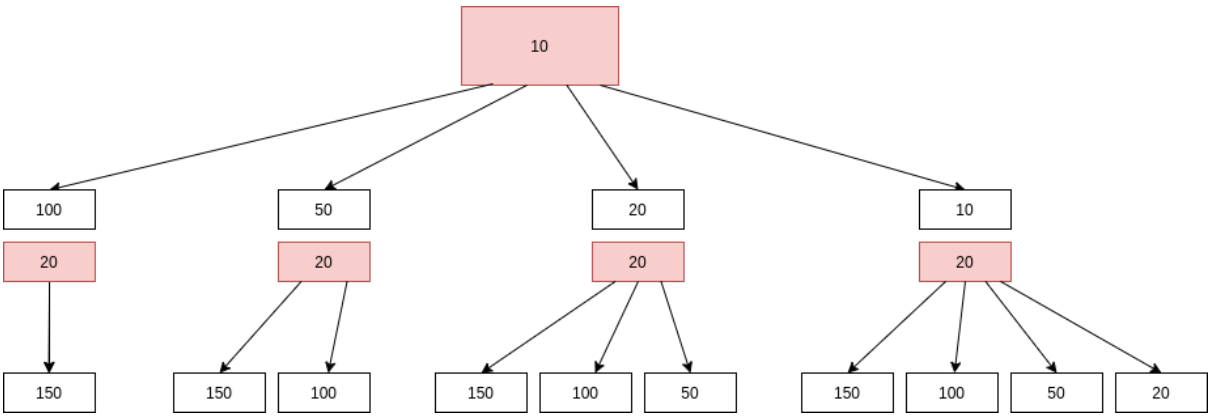
É de referir também que existiu a restrição de que não poderiam existir situações em que um barco com menor rating ficasse atribuído a um porto com maior quantidade de peixe em comparação com um barco com maior rating.

Construção e descrição do algoritmo

Dado às características do problema, conclui-se que a sua resolução poderia ser construída através de uma abordagem de programação dinâmica em detrimento de outras como greedy (pois a melhor solução em cada momento poderia levar a soluções incorretas) ou força bruta (o número de possibilidades torna esta abordagem inviável).

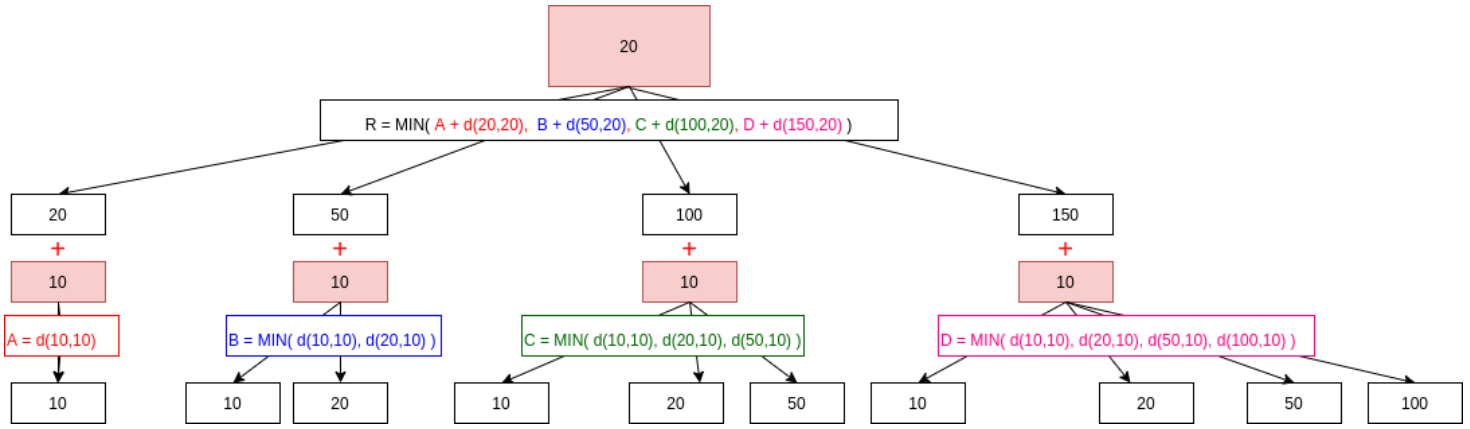
Por isso, começou-se por caracterizar uma solução ótima recorrendo ao esquema seguinte que representa um possível cenário e todas as suas soluções possíveis.





É possível observar para os portos 20 e 10 (valores correspondentes à quantidade de peixe) que as soluções se repetem, pelo que apenas é necessário considerar uma das subárvores correspondentes a um porto para encontrar a solução.

Focando-se então na subárvore correspondente ao porto 20, a solução ótima para este subproblema é apresentada em baixo.



Nota: **d(i,j)** corresponde à distância do barco i ao porto j

Sendo que o número de distâncias necessárias a calcular por porto em função do número de barcos (nb) e do número de portos (np) é dado por:

$$f(nb, bp) = (nb - np) + 1$$

para o cálculo iterativo pretendido quando **nb ≥ np**, pensou-se no tabelamento das distâncias da seguinte forma:

		Barcos				
		10	**20**	**50**	**100**	**150**
Portos	**10**	D10	D11	D12	D13	X
	20	X	D21+D10	D22+D11	D23+D12	D24+D13

X: distâncias que não são necessárias de calcular por se referirem a situações impossíveis.

D10 = d(10,10)
D11 = min(D10, d(20,10))
D12 = min(D11, d(50,10))
D13 = min(D12, d(100,10))

D21 + D10 = d(20,20) + D10
D22 + D11 = d(50,20) + D11(=min(D10,D11))
D23 + D12 = d(100,20) + D12(=min(D10,D11,D12))
D24 + D13 = d(150, 20) + D13(=min(D10,D11,D12,D13))

Considerando o array ***distancias*** representativo da tabela acima, o resultado da distância ótima encontra-se em ***distancias[np-1, nb-1]***.

Para o total de ratings dos barcos, o tabelamento é idêntico ao das distâncias, por isso, sendo este representado pelo o array ***ratings***, o resultado encontra-se em ***ratings[np-1, nb-1]*.**

O número total de peixe capturado é a soma da quantidade de peixe existente em todos os portos.

Já quando **nb < np**, o total de peixe capturado, o total da distância e o total dos ratings pode ser calculado em função dos barcos (b) e portos (s) através:

$$r(b,s) = \sum_{j=np-nb}^{np} \sum_{i=0}^{nb} (s[j].value,\ d(b[i], s[j]),\ b[i].value)$$

Análises das complexidades temporal e espacial

Complexidade temporal

Para o cálculo da complexidade temporal da função **store_info():**

As instruções de complexidade temporal constante, $O(1)$, como é o caso das afetações que são executadas n vezes. O sorting do array de objetos por se tratar de um mergesort tem como complexidade temporal $O(n\ log(n))$. Logo, a complexidade temporal da função é dada por:

$$n\ *\ O(1)\ +\ O(n\ log(n))\ =\ O(n)$$

Para o cálculo da complexidade temporal da função **resolve()**:

As instruções de complexidade temporal constante, $O(1)$, como é o caso das comparações, dos saltos condicionais, das incrementações e das afetações, são executadas para o pior caso (nb = 2 * np e nb máximo) um total de n / 2 * n vezes, ou seja, a complexidade temporal da função é dada por:

$$[(n\ /\ 2)\ *\ n]\ *\ O(1)\ =\ O(n^2)$$

A complexidade temporal das funções **calculate_distance()** e **main()** é de $O(1)$.

Complexidade espacial

Para o cálculo da complexidade espacial da função **store_info()**:

Variáveis do tipo **int**: 4 * 1 = 4 B -> $O(1)$
Variável **String** -> $O(n)$

Info[ ] boats: 4 * 3 * 4000 = 48 000 B - > $O(n)$
Info[ ] spots: 4 * 3 * 4000 = 48 000 B -> $O(n)$

O sorting por se tratar de um mergesort -> $O(n)$

Para o cálculo da complexidade espacial da função **resolve()**:

Variáveis do tipo **int**: 4 * 10 = 40 B -> $O(1)$

int[ ][ ] distances: 4 * 4000 * 4000 = 64 000 000 B -> $O(n^2)$
int[ ][ ] ratings**:** 4 * 4000 * 4000 = 64 000 000 B -> $O(n^2)$
int[ ] output: 4 * 3 = 12 B -> $O(n)$

Chamada da função **calculate_distance()**: 4*4000 = 16 000 B -> $O(1)$ * n = $O(n)$

Para o cálculo da complexidade espacial da função **main()**:

Variáveis do tipo **int**: 4 * 2 = 8 B -> $O(1)$
Variável **String** ->$O(1)$
Variável **BufferedReader** -> $O(1)$
Chamadas das funções **store_info()** e **resolve()** -> $O(1)$