1 Apresentação do Problema

O problema apresentado consiste num robot, denominado *Robotruck* que tem de entregar um conjunto de pacotes aos seus respectivos destinos numa fábrica. As encomendas têm de ser entregues pela ordem de chegada à central de distribuição. Os destinos de entrega são representados através de coordenadas cartesianas a duas dimensões e a distância entre dois pontos de entrega corresponde à *Distância Manhattan* entre eles, ou seja, o módulo da diferença das abcissas somado ao módulo da diferença das ordenadas entre os dois pontos.

Todos os pacotes têm um certo peso, que nunca é superior à capacidade máxima do robot. O *Robotruck* tem uma capacidade de transporte limitada, não sendo por isso possível carregar todos os pacotes de uma vez.

O desafio proposto pelo problema consiste portanto em implementar um algoritmo que permita calcular a distância mínima que o robot precisa de percorrer para entregar todos os pacotes. $\boldsymbol{\theta}$

2 Resolução do Problema

Implementámos diversos algoritmos para a resolução do problema proposto. A primeira aproximação correspondeu a um algoritmo de complexidade O(n^3), que foi rejeitado pelo Mooshak com *Time Limit Exceeded*.

Implementámos ainda um outro algoritmo, recorrendo a uma diminuição do número de cálculos de pesos e distâncias, dessa forma obtivémos uma complexidade de O(n^2) para o algoritmo. Este algoritmo foi também rejeitado pelo Mooshak com *Time Limit Exceeded*.

Por fim conseguimos definir uma expressão recursiva de apenas um argumento e que apenas utilizava um *for* para invocar as chamadas recursivas necessárias. Este algoritmo utiliza a noção de peso e distâncias acumulados para evitar o uso de mais *fors*. Desta forma conseguimos obter um algoritmo de complexidade O(n*c) e que foi aceite pelo mooshak.

3 Implementação do Algoritmo

A técnica de desenho de algoritmos utilizada para resolver o problema foi a técnica da programação dinâmica. Começámos por definir a seguinte função recursiva para formalizar o algoritmo:

$$D(i) = \begin{cases} cost(origin, 1) + d(1, i) + cost(i, origin) & \text{if } w(1, i) \leq Capacity, \\ \min_{l=0}^{k}(cost(Origin, i-l) + d(i-l+1, i) + cost(i, Origin) + D(i-l-1)) & \text{if } w(1, i) > Capacity. \end{cases}$$

Descrição:

- D(i): distância mínima para entregar os pacotes de 1 a i
- d(i,j): distância acumulada entre os pontos de i a j
- cost(i,j): distância Manhattan desde o ponto i ao ponto j
- w(i,j): peso acumulado dos pacotes desde i a j
- k: valor máximo de l tal que w(1,i+l) é menor ou igual à capacidade do robot

De seguida passámos à fase de implementação das classes e métodos necessários para a implementação da versão dinâmica do algoritmo. Durante a leitura dos dados da consola é registado em dois arrays distintos o peso e a distância acumulados até à posição do pacote lido da consola. Estes dados servem para auxiliar o cálculo da distância e peso contidos entre qualquer intervalo $[i,j]$ de pacotes. Estes dados são fundamentais para a baixa complexidade temporal do algoritmo.

3.1 Classes

As classes que constituem o nosso programa são:

- Main: classe onde o algoritmo está implementado, bem como os seus métodos e estruturas de dados auxiliares
- Package: classe utilizada para guardar os dados relativos a um pacote

Os dados armazenados na classe *Package* são:

- Coordenada x do destino
- Coordenada y do destino

Estes pacotes são guardados num array de pacotes à medida que os dados para a criação de um são lidos da consola.

3.2 Métodos

Os Métodos implementados são os seguintes:

- cost(int x1, int y1, int x2, int y2): calcula a distância Manhattan entre as coordenadas dadas como argumento
- w(int i, int j): calcula o peso acumulado dos pacotes desde i a j recorrendo aos dados armazenados no vector de pesos acumulados
- d(int i, int j): calcula a distância acumulada dos pacotes desde i a j recorrendo aos dados armazenados no vector das distâncias acumuladas
- D(int i): metodo que executa o algoritmo, calcula a distância mínima para entregar os pacotes desde 1 até i

3.3 Estruturas de dados

As estruturas de dados utilizadas para implementar o algoritmo consistem simplesmente em arrays. São utilizados quatro arrays no algoritmo, três deles são utilizados para reduzir a complexidade do algoritmo para O(n*c) e o último é necessário para implementar a versão dinâmica do algoritmo.

Utilizámos arrays simples pois os diversos passos do algoritmo sabem exactamente que posições dos arrays é que precisam de indexar, e os arrays permitem acessos por posição com complexidade O(1). Durante a execução do algoritmo nunca ocorrem remoções nos arrays e e as inserções são sempre feitas à cabeça dos arrays, e portanto, com complexidade O(1).

Está implícito que cada posição do array corresponde ao número do pacote. Desta forma o algoritmo consegue efectuar todas as operações de que necessita sobre as estruturas de dados auxiliares com complexidade O(1).

4 Análise do Algoritmo

O algoritmo implementado utiliza quatro arrays de tamanho n, onde n é o número de pacotes. Pode-se assim concluir que a complexidade espacial do algoritmo é O(4n), que pertence à categoria de O(n), ou seja, linear. Temos portanto a memória utilizada pelo algoritmo a variar linearmente com o número de pacotes a entregar.

Quanto à complexidade temporal decidimos efectuar alguns cálculos para determinar a sua grandeza. Temos que

$$\sum_{i=1}^{n}\sum_{l=0}^{k} 1$$

$$\sum_{i=1}^{n} k + 1$$

Temos que k é menor ou igual à capacidade, sendo que no caso de os pacotes terem todos peso um, teriamos k igual à capacidade e portanto

$$\sum_{i=1}^{n} C + 1$$

Assim obtemos como complexidade temporal do algoritmo O(n*C + n), que corresponde à categoria de complexidade O(n*C).

5 Conclusões

A nossa implementação tem como pontos fortes o facto de calcular bastantes dados necessários ao algoritmo em tempo de leitura de dados da consola, algo que é bastante bom pois efectua operações numa fase indispensável para a execução do algoritmo. Tal propriedade permite a baixa complexidade do algoritmo.

Apesar deste ponto forte da implementação é bastante provável que existam formas mais eficientes de calcular os dados necessários para cálculo de pesos e distâncias acumuladas. Esta é uma alternativa que merecia ser estudada, e se tal for possível, seria um ponto fraco para o nosso algoritmo necessitar de e struturas de dados de tamanho n para armazenar dados acumulados que poderiam ser calculados com um menor consumo de memória.

Apesar de todos estes prós e contras, a solução implementada apresenta uma complexidade temporal que é muito provavelmente da mesma ordem de grandeza que a solução optimal para este problema, residindo as maiores dúvidas na complexidade espacial do algoritmo.