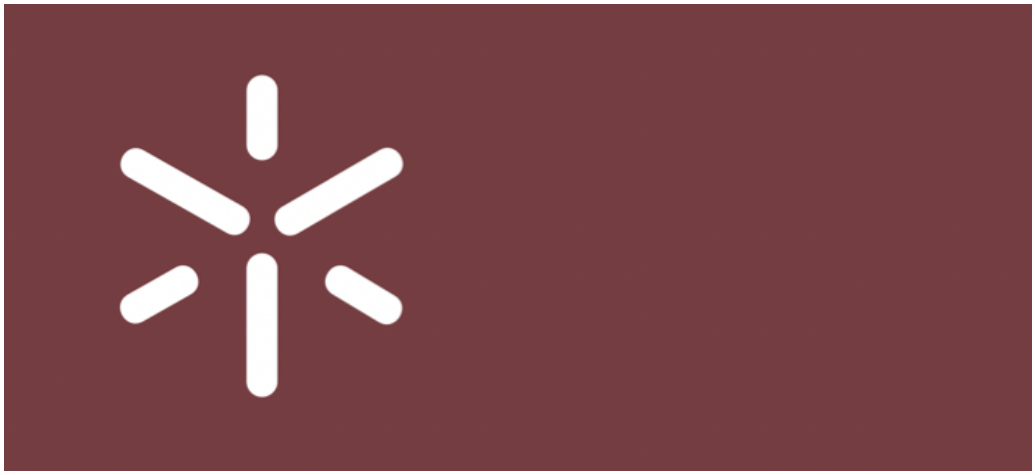

UNIVERSIDADE DO MINHO

Arquitetura e Cálculo

TP2 - Modelling and analysis of cyber-physical systemsnow with *monads*

Rafael Lourenço (A86266)

Rodolfo Silva (A81716)

Conteúdo

1 Introdução

Este relatório é redigido no âmbito da UC de Arquitetura e Cálculo do perfil de mestrado MFES do curso de Engenharia Informática da Universidade do Minho. Ao longo das próximas secções, será descrito alguns exercícios propostos e as suas resoluções, o problema proposto sendo este baseado no caixeiro viajante, e o código em Haskell, com o uso de *monads*, desenvolvido para o resolver.

2 1º Parte

2.1 Exercício 1

Neste exercício foi-nos proposto calcular o *output* do seguinte *lambda* termo para o *input* 0:

1 $$x : \mathbb{N} \vdash_c y \leftarrow choice(return(x-1), return(x+1)); choice(return(y-1), return(y+1)) : \mathbb{N}$$

Com isto em mente, é importante realçar certos pontos. A função *choice* recebe um par de duas listas, retornando a junção das duas. Neste caso, a função *return* irá colocar o valor de `x-1` numa lista singular e o mesmo acontece com o `x+1`. Esta função pode ser vista como um construtor de *monads*, uma vez que, coloca um certo valor dentro de um *monad* (neste caso no *monad* das listas, devido à função *choice*). A variável y pode ser visto como um iterador, que irá percorrer o resultado da função *choice*.

Dado isto, para o input igual a **0**, em primeiro lugar irá realizar `choice(return 0-1 , return 0+1) =[-1,1]`. Logo o y irá ter estes dois valores:

1. Na primeira iteração, irá realizar o `choice( -1-1 , -1+1 )` e retornará a lista = [-2,0].
2. Na segunda iteração, irá realizar o `choice( 1-1 , 1+1 )` e retornará a lista = [0,2].

A estes dois outputs é aplicado o *concat* , que é feito internamente pela operação *bind* , resultando assim no output **[-2,0,0,2]**.

2.2 Exercício 2

Neste exercício foi pedido para desenvolver um programa monádico que calculasse os ciclos hamiltonianos de um grafo. Um ciclo hamiltoniano consiste num caminho que permite passar por todos os vértices de um grafo **apenas uma vez**, sendo também necessário terminar esse caminho exactamente no vértice inicial.

Deste modo, os vértices são representados pelo tipo de dados *Node*, sendo que cada aresta é representada por um par de *Nodes*. Assim, foi construída a representação de um grafo através de uma função, com a seguinte parametrização `adj :: (Node,Node) -> Bool`. Através desta função consegue-se verificar se um par de vértices são adjacentes.

Também foram usadas várias funções auxiliares, tais como:

- `adjacentNodes :: Node -> [Node] -> [Node]` - Esta função calcula todos os vértices adjacentes a um vértice, retornando uma lista com os mesmos.
- `allNodes :: [Node]` - Esta função calcula todos os vértices do grafo.
- `choice :: ([a],[a]) -> [a]` - Esta função faz a junção das duas listas passadas como parâmetro no par, também usada no exercício 1.

A partir destas funções apresentadas anteriormente foi desenvolvida a seguinte função:

```
addtoEnd :: Path -> [Node] -> [Path]
addtoEnd p ns = let ns_filtered= ns \\ p in
    (\x -> p++[x]) <$> ns_filtered
```

Esta função serve para calcular o próximo vértice de um caminho já existente. Isto é possível, calculando todos os vértices adjacentes ao último vértice do caminho existente, sendo que estes não podem pertencer ao caminho.De seguida, a partir dessa lista, são calculados todos os caminhos possíveis adicionando um desses ao final do caminho existente.Todos os novos caminhos resultantes são constituídos por mais um vértice que o inicial(passado como parâmetro).

Após a resolução desta função foi iniciada a resolução da função principal, a `hCycles`, que tem como objetivo o calculo de todos os ciclos hamiltonianos. Apresenta-se de seguida a sua implementação:

```
hCycles :: Node -> [Path]
hCycles n = do
        c1 <- (addtoEnd [n] $ adjacentNodes n allNodes)
        c2 <- (addtoEnd c1 $ adjacentNodes (last c1) allNodes)
        c3 <- (addtoEnd c2 $ adjacentNodes (last c2) allNodes)
        c4 <- (addtoEnd c3 $ adjacentNodes (last c3) allNodes)
        c5 <- (addtoEnd c4 $ adjacentNodes (last c4) allNodes)
        c6 <- rem_non_cycles c5
        return c6
```

Esta função recebe um vértice inicial, que representa o ponto inicial da travessia do grafo.Inicialmente, esse vértice é colocado numa lista e a partir deste calcula-se todas os os vértices adjacentes, através da função `addtoEnd` que foi explicada anteriormente. Deste modo, em c1 estão todos os caminhos possíveis com 2 vértices, em c2 estão todos os caminhos possíveis com 3 vértices e assim sucessivamente até c5. Em C5, temos então todos os caminhos com 6 vértices, e dado que, o número de vértices do grafo é seis, neste momento já temos todos caminhos hamiltonianos possíveis, todavia alguns não são ciclos, pois o vértice não está adjacente ao nó inicial. Para isso, foi criada a função `rem_non_cycles`,que verifica se um caminho hamiltoniano pode formar um ciclo hamiltoniano, como se pode ver na sua definição:

```
rem_non_cycles::Path->[Path]
rem_non_cycles [] = []
rem_non_cycles path
  | elem (head path) (adjacentNodes (last path) allNodes) = [ choice (path,[(head
  path)])]
  | otherwise = []
```

É de realçar que nestas funções estamos a tirar partido da notação e operadores monádicos, uma vez que, estamos a usar o operador bind para aplicar uma uma espécie de (concat . map) à estrutura, que neste caso são listas. Praticamente, esta operação está escondida com a *syntax* do **do**, pois permite uma melhor legibilidade, uma vez que, `c1 <- addtoEnd (...)` é equivalente a `addtoEnd (...) >>= ( \ x ->` .

3 2º Parte

3.1 Exercício 3

Árvore de derivação termo 1:

$$\dfrac{\dfrac{\dfrac{[\![x:\mathbb{A}, y:\mathbb{A}\to\mathbb{A}\vdash_c y:\mathbb{A}\to\mathbb{A}]\!]\quad [\![x:\mathbb{A}, y:\mathbb{A}\to\mathbb{A}\vdash_c x:\mathbb{A}]\!]}{(wait_2,1)[\![x:\mathbb{A}, y:\mathbb{A}\to\mathbb{A}\vdash_c y\ x:\mathbb{A}]\!]}\quad (wait_1,return\ 1)[\![x:\mathbb{A}, z:\mathbb{A}\vdash_c z:\mathbb{A}]\!]}{(wait_1,1)[\![x:\mathbb{A}, z:\mathbb{A}\vdash_c return\ z:\mathbb{A}]\!]}}{[\![x:\mathbb{A}, y:\mathbb{A}\to\mathbb{A}\vdash_c wait_2(yx):\mathbb{A}]\!]}\quad [\![x:\mathbb{A}, z:\mathbb{A}\vdash_c wait_1(return\ z):\mathbb{A}]\!]}{[\![x:\mathbb{A}\vdash\lambda y:\mathbb{A}\to\mathbb{A}.wait_2(yx):\mathbb{A}\to\mathbb{A}]\!]}\quad [\![x:\mathbb{A}\vdash\lambda z:\mathbb{A}.wait_1(return\ z):\mathbb{A}]\!]}{[\![x:\mathbb{A}\vdash_c(\lambda y:\mathbb{A}\to\mathbb{A}.\ wait_2(y\ x))(\lambda z:\mathbb{A}.\ wait_1(return\ z)):\mathbb{A}]\!]}$$

Árvore de derivação termo 2:

$$\dfrac{\dfrac{(\mathrm{wait\_3}, \mathrm{return}\ \ 1)[x:\mathbb{A}\vdash_c x:\mathbb{A}]}{(\mathrm{wait\_3}, 1)[\![x:\mathbb{A}\vdash_c return\ x:\mathbb{A}]\!]}}{[\![x:\mathbb{A}\vdash_c wait_3(return\ x):\mathbb{A}]\!]}$$

Visto que y é uma função de $\mathbb{A}\to\mathbb{A}$ esta não vai alterar o x logo vamos ter que $y(x)=x$. Assim têm-se $(wait_1, return z)+(wait_2, x)$ sendo x o output da primeira parte e input da segunda parte, ou seja o resultado final vai ser $(wait_3, return\ x)$ o que é equivalente ao output do segundo *lambda* termo $(wait_3, return\ x)$.

3.2 Exercício 4

Este exercício introduz o problema do caixeiro viajante, onde o vendedor tem de percorrer varias cidades, de forma a, realizar as suas entregas percorrendo o menor caminho possível, reduzindo assim, o tempo necessário para a viagem e os possíveis custos com transporte e combustível.

Normalmente, para a representação deste problema utilizam-se os grafos onde cada vértice é uma cidade e uma aresta a distância/tempo entre duas cidades. De certa forma, este exercício é uma evolução do exercício 2, com a introdução de novas nuances.

A primeira mudança é na representação das arestas, visto que no exercício 2 as arestas não eram pesadas e apenas era necessário saber se existia ligação ou não. Para a implementação das arestas pesadas foi usado o *monad Maybe*, visto que, duas cidades podem estar ligadas através de uma aresta ou não, e no caso de estarem, estas permanecem a uma distância fixa entre si.A função passou a ter a seguinte parametrização de tipos `adjT :: (Node,Node) -> Maybe Int`, onde praticante um par de Nodes/cidades a uma distancia de 10 unidades,é representado por `Just 10` . Caso não estejam ligadas, a função retorna o valor `Nothing`.

Outras mudanças significativas, foi nas funções auxiliares, sendo que, agora estas fazem uso do *monad Duration*, que servirá para acumular o tempo das viagens entre cidades/vértices.

O *monad Duration* associa um inteiro ao tipo que recebe como parâmetro, onde este inteiro tem uma representação temporal.A definição do *monad* é feita da seguinte forma:

```
data Duration a = Duration (Int, a) deriving (Show,Eq)

getDuration :: Duration a -> Int
```

```
getDuration (Duration (d,x)) = d

getValue :: Duration a -> a
getValue (Duration (d,x)) = x

instance Monad Duration where
    (Duration (i,x)) >>= k = Duration (i + (getDuration (k x)), getValue (k x))
    return x = (Duration (0,x))
```

Praticamente temos de definir como o operador bind e a função return funcionam.No caso do return é retornado o estado inicial, ou seja, com a variável "temporal"a 0. No caso do operador *bind* (>>=) , o comportamento definido é que caso um função k seja aplicada ao *(Duration (i,x))*, este ira executar a função ao tipo de dados, somando o tempo(i) com o resultado obtido. Os *monads* também usam outras instâncias, tais como o *Functor*, que serve para nos mostrar como se aplicaria uma função ao tipo de dados, e a *Applicative* que nos permite executar uma função, que esteja dentro do *monad Duration*, ao tipo de dados *Duration*, como se pode verificar no excerto de código apresentado:

```
instance Functor Duration where
  fmap f (Duration (i,x)) = Duration (i,f x)

instance Applicative Duration where
  pure x = (Duration (0,x))
  (Duration (i,f)) <*> (Duration (j, x)) = (Duration (i+j, f x))
```

Ainda nesta modulo foram usadas algumas funções auxiliares, como por exemplo, obter o par (Int,a) de Duration(Int,a), a função `wait` , que permite adicionar mais tempo à variavel temporal (Int) etc.
Passando então a definição do algoritmo, similarmente ao exercicio 2 existem duas funções principais:

- `taddToEnd ::  Duration Path -> [Duration Node] -> [Duration Path]`

  O comportamento implementado nesta função é similar à função addToEnd , onde são calculadas todas as possibilidades dado um caminho inicial. A diferença é que usamos o *monad* da duração e em vez de ter apenas uma lista de nós, usamos uma lista de *Monads Duration* para representar os tais tempos já explicados anteriormente. A implementação passsa por aplicar um filtro todos os nos que não pertençam ao caminho existente, sendo que, posteriormente esses serão adicionados ao caminho um a um, formando vários novos caminhos, como podemos ver no codigo apresentado de seguida:

  ```
  taddToEnd :: Duration Path -> [Duration Node] -> [Duration Path]
  taddToEnd p ns = do
          s1 <- filter (\ (d,v) -> not (elem v (getValue p) ) )  (getPair <$> ns)
          f <- (\ (d,v) -> Duration(getDuration p + d,(getValue p) ++ [v]))  <$> [s1];
          return f
  ```

  É de realçar que o funcionamento do operador $< \$ >$ é similar a um map, sendo que `(getPair <$> ns)` é similar a `(map getPair ns)` . Este operador usa a instância do *functor* mostrada acima.

- `hCyclesCost ::  Node -> [Duration Path]`

Quanto à função principal, esta também é semelhante à hcycles do exercício 2, apresentando exatamente o mesmo comporatamento. Também é usada a função `rem_non_cycles2` para remover os caminhos dos quais não é possivel formar um ciclo hamiltoniano. A implementação das funções apresentam-se de seguida:

```
hCyclesCost :: Node -> [Duration Path]
hCyclesCost n = do
   c1 <- (taddToEnd (Duration(0,[n])) $ tadjacentNodes n allNodes) ;
   c2 <- taddToEnd c1 $ tadjacentNodes  ((last . getValue ) c1 ) allNodes;
   c3 <- taddToEnd c2 $ tadjacentNodes  ((last . getValue ) c2 ) allNodes;
   c4 <- taddToEnd c3 $ tadjacentNodes  ((last . getValue ) c3 ) allNodes;
   c5 <- taddToEnd c4 $ tadjacentNodes  ((last . getValue ) c4 ) allNodes;
   c6 <- rem_non_cycles2 c5
   return c6

catmaybes =  (\x -> case x of {Just y -> [y]; _ ->[]})

rem_non_cycles2::Duration Path->[Duration Path]
rem_non_cycles2 (Duration (d,path))
 | elem (head path) (adjacentNodes (last path) allNodes) =
    let v =  ((head.catmaybes .adjT)  (last path,head path))
        in  [ wait v  (Duration (d,path++[(head path)]))]
 | otherwise = []
```

De forma a concluir o algoritmo, é calculado a rota que apresenta o menor tempo de viagem, e para isso foi usada a seguinte função:

```
tsp = minimum . hCyclesCost
```

Esta função pode ser aplicado qualquer um dos nós existentes no grafo. Assim, caso queiramos garantir que dá sempre o mesmo ciclo começando em qualquer nó/cidade, basta executar o seguinte código:

```
tsp <$> allNodes
```

4 3º Parte

4.1 Exercício 5

O uso de *monads* em programação funcional traz muitas vantagens. Estes permitem uma simplificação do código através da redução da quantidade de código repetido que fica implícito com o uso de *monads*. Por exemplo, na criação de parsers, *monads* podem ser usados para *error handling* e assim evitamos de estar sempre a escrever testes explicitos para casos onde falha, uma vantagem de *monads* nestes casos é que os efeitos secundários não podem escapar o contexto monádico.

Outra vantagem do uso de *monads* seria o facto de que possibilitam a definição de novas funcionalidades na línguagem que executam diversas funcionalidades, tal como, manipulações ao estado interno como externo de um programa, e consequentemente permitirem comunicações com outras línguagens.

Monads também possuem bastante flexibilidade e podem ser usados para fazer novos *monads* que os incorporem e sejam mais complexos. Parte desta flexibilidade vem também, do facto de, poderem tirar partido da composição de funções e várias leis associadas à mesma.

Quanto às desvantagens pertencentes a *monads*, estas debruçam-se mais sobre dificuldades que podem ser encontradas por parte de um utilizador e não por lados negativos do uso de *monads* em si. O uso de *monads* requer um conhecimento teórico mais aprofundado por parte do utilizador, e visto que, muitos dos tutoriais de *monads* que existem são de certa forma maus[4], esta torna-se numa tarefa complicada de aprender.

Também por parte de consequência da desvantagem anterior, o uso de *monads* leva de certa forma uma adaptação custosa e uma aprendizagem lenta a como os usar e aplicar. Em parte, devido ao facto de que, pelo menos na linguagem Haskell, operadores monádicos costumam ser representados por símbolos o que leva a uma compreensão mais lenta do código por parte do utilizador.

5 4º Parte

Esta parte do projeto refere-se a uma secção mais exploratória do trabalho, onde o grupo decidia algumas funcionalidades extras para implementar nos problemas já explicados anteriormente. De acordo com a sugestão dos docentes, decidimos implementar a leitura dos grafos de um ficheiro.Também implementamos funções para escrita dos dois tipos de grafos existentes em ficheiros.Estas implementações serão explicadas detalhadamente na secção do exercício 6.1.

Por último, dado que, o mundo dos veículos terrestres tende a tornar-se totalmente elétrico e autónomo ao longo dos próximos anos, decidimos implementar o comportamento de um veículo desses. Para isso, foi usado o *monad state* que permite simular o carregamento e descarregamento do veículo. A ideia passa por ter um veículo que pretende passar por vários pontos de referencia(todos os nos do grafo) para deixar uma ou mais encomendas.O objetivo do algoritmo passa por otimizar a bateria do veículo, maximizando a vitalidade das baterias e minimizando o custo que se teria, por o carro estar a ser carregado ao longo da travessia. Este exercício será explicado mais detalhadamente na secção do exercício 6.2.

5.1 Exercício 6.1

Começando por explicar como foi feita a escrita dos grafos para ficheiros,implementamos um função capaz de gerar todas as combinações possíveis de pares de vértices, como se mostra no seguinte excerto:

```
pairs :: [a] -> [(a, a)]
pairs l = [(x,y) | x <-l, y<-  l]
```

De seguida, foi então implementada duas funções capazes de tranformar as funções `adj` e `adjT` em strings, utilizando syntax monadica, como se pode ver abaixo:

```
graphToStr = do
                p <-pairs allNodes
                p2 <- filter (\x -> adj x) [p]
                str <- toStr p
                return str


graphToStr2 = do
                p <-pairs allNodes
                p2 <- filter (\x -> case (adjT x) of {Just y -> True; _ ->False}) [p]
                str <- toStr2 p
                return str
```

Por último, utilizando o monad IO do haskell e a função auxliar `writeFile` que escreve uma string para um ficheiro, criamos a seguinte função:

```
graphToFile :: FilePath -> Bool -> IO ()
graphToFile filename  bool
    | bool = writeFile filename graphToStr2
    | otherwise  = writeFile filename graphToStr
```

Praticamente, esta função recebe o nome do ficheiro para onde se pretende escrever o grafo, e um booleano que é usado como `True` para escrever o grafo pesado usado no exercício 4 e como `False` para o grafo do exercício 2.

Passando agora a explicar a leitura dos ficheiros,é de realçar que, esta permite ler os ficheiros escritos pelas funções explicadas anteriormente. Caso se queira criar os ficheiro manualmente, estes apresentam o seguinte formato:

```
A B
A C
A F
B A
B C
B E
B F
C A
C B
C D
D C
D E
E B
E D
E F
F A
F B
F E
```

Figura 1: Exemplo da *syntax* do ficheiro para grafos **não pesados**

```
A B 2
A C 3
A F 6
B A 30
B C 0
B E 4
B F 3
C A 60
C B 3
C D 50
D C 2
D E 3
E B 1
E D 3
E F 2
F A 4
F B 5
F E 3
```

Figura 2: Exemplo da *syntax* do ficheiro para grafos **pesados**

Deste modo, poder realizar a leitura destes ficheiros para uma estrutura de dados, foi criado o seguinte tipo de dados :

```
type Grafo = [(Node,Node,Int)]
```

De forma a, tirar partido do monad IO aqui foram usadas várias funções auxiliares já criadas nas bibliotecas do haskell, tais como :

- `openFile` - abre um ficheiro, retornando o seu descritor .
- `hGetContents` - Obtém o conteúdo de um ficheiro para um string ,retornado dentro do monad IO .
- `hClose` - fecha o descritor do ficheiro.
- `words` e `lines` - uma responsável por separar string por espaços e outra responsável por separar as strings por "$\backslash n$"

Ainda foram implementadas várias funções auxiliares para realização de *cast's* em *haskell*, todavia, as suas definições serão omitidas ,devido a, apresentarem baixa complexidade.Assim, mostra-se de seguida, a função principal responsável por ler estes dois tipos de ficheiros.

```haskell
readFromFile :: FilePath -> IO ()
readFromFile filename  = do
            f <- openFile filename ReadMode  -- Abre o descritor para leitura
            contents <- hGetContents f
            let g = (words <$> (lines contents) ) in
                let grafo =  toGraph ((length . head) g)  g in
                    print grafo
            hClose f                         -- Fecha o descritor
```

A função apresentada acima, começa por abrir o ficheiro em modo de leitura, passando, de seguida, a obter o conteúdo da mesma. Posteriormente, essa string é separada por linhas,sendo que, cada linha é separada por espaços, colocando em g uma `[[String]]`. Após isto , é aplicada a função `toGraph` que realiza o *cast* dessa lista de listas para o tipo de dados `Grafo`. Por fim, é fechado o descritor do ficheiro aberto. Para usar este grafo bastava passar como parâmetro às funções, onde realizamos o `print grafo`.

5.2 Exercício 6.2

Devido ao exercício 6.2 ter sido realizado em primeiro lugar, as funções usadas para a representação de grafos são as do exercício 4. No entanto, para o usar a representação em ficheiros, a forma mais simples seria passar o grafo por parâmetro,como foi dito anteriormente. Esta mudança não seria complicada de implementar, porém não achamos necessário fazer essa alteração.

Deste modo, para este exercício os pesos do grafo passam a ser **distâncias**(Km), entre 2 pontos no mapa/grafo. O veículo elétrico pode ter várias configurações a nível do consumo e bateria inicial.Para isso são usadas duas variáveis, como é mostrado abaixo:

```haskell
-- Configs do veículo elétrico.
type Battery = Int
consuption_KM = 1   -- Consumo do veículo por km;
batery_state::Int
batery_state  = 40 -- Estado inicial da bateria;
```

Tanto o consumo como a bateria inicial estão em percentagens, ou seja , o veículo começará com 40 % de bateria e a cada quilometro tem um consumo de 1 % de bateria. O carregamento do veículo não pode ser executado em qualquer momento e também existe um limite de percentagem por cidade, representando assim o tempo que se pode permanecer em cada cidade.Isto é implementado com o *monad Maybe* e através da seguinte função:

```haskell
charging_stations :: Node -> Maybe Int
charging_stations p = case p of
    A -> Just 10
    C -> Just 30 -- é possivel abastecer num máximo até 30 \% da bateria.
    F -> Just 70
    _ -> Nothing
```

Como podemos ver no excerto acima, no ponto A podemos carregar no máximo 10 % de bateria, no ponto C podemos carregar 30 % e por ultimo no ponto F podemos carregar 70%, sendo que em todos os outros não é possível fazer o carregamento.

De forma a, simular a deslocação do veículo foi usado o monad State, pois permite ter uma noção do estada da bateria. Para a implementação deste comportamento, foram usadas as seguintes funções:

```
--Muda o estado da bateria somando o inteiro passado.
change_bat::Int -> State Battery ()
change_bat v =  State $ \x -> ((),normalize  (x+v ))


-- Coloca o valor do estado
tos :: State Battery Int
tos = State $ \x-> (x,x)
```

Através da função change_bat, tanto é possivel aumentar a bateria como diminuir a mesma. A função tos é responsavel por alterar o tipo do monad de `State Battery ()` para `State Battery Int`, colocando o ultimo valor da bateria no tal inteiro.

```
--teste para a função changebat
test :: State Battery Int
test = do
    change_bat 10
    change_bat 10
    change_bat $ -5
    tos


-- evalState2 test batery_state
```

Por fim a função `test` é uma demonstração como estas funções poderiam ser usadas. Para testar basta executar o a linha que está em comentário.

O algoritmo para realização da travessia, consiste essencialmente em 4 passos:

1. São calculados todos os ciclos hamiltonianos, com o auxílio da função `hCycles` do Exercício 2
2. Calcular a bateria necessária em cada nodo e todas as possibilidades de carregamento.
3. A partir do ponto 2, calcular a bateria atual em cada nó.
4. Detetar e remover todas as rotas que contêm, em algum momento, a bateria igual 0.
5. Calcular o caminho que minimiza o tempo de carregamento, para todos os ciclos hamiltonianos.

O calculo da bateria necessária é uma multiplicação do consumo por km pela distancia e está implementado na seguinte função:

```
calcula_bat_necessaria::[Node] -> [Int]
calcula_bat_necessaria []= []
calcula_bat_necessaria [a] = []
calcula_bat_necessaria (x:x2:xs) = ((\p -> case p of {Just y -> y* consuption_KM; _ ->0})
                                    (adjT (x,x2) )
                                    ): calcula_bat_necessaria (x2:xs)
```

De seguida é feito um calculo para a bateria que poderá ser carregada.Praticamente, dado um conjunto de nós , esta função irá calcular a percentagem que se poderá calcula nessa cidade, podendo atingir valores entre 0 a 100%.

```
calcula_chargingPoint::[Node]->[Int]
calcula_chargingPoint []= []
calcula_chargingPoint [a] = []
calcula_chargingPoint (x:xs) = ((\p -> case p of {Just y -> y ; _ ->0})
                                (charging_stations x )
                               ): calcula_chargingPoint xs
```

A partir desta função é calcula a todas as combinações de possíveis carregamentos,como por exemplo,no caso de fazer a seguinte viagem [A,B,C,A], este pode carregar no ponto A e C. Logo a lista de combinações possíveis seria algo como [[10,0,30],[10,0,0],[0,0,30]], ou seja, ou carrega nos dois pontos A e C, ou apenas num deles

Posteriormente a este passo, é então calculada para cada rota as possíveis configurações de carregamento e a bateria atual em cada momento(nó) da viagem. É de realçar que isto é calculado com um subtração normalizada entre as duas ultimas funções explicadas anteriormente.A função responsável por este comportamento é a seguinte:

```
apply l =  do {   y<- (allpossibilities. calcula_chargingPoint) l ;
                return (zipWith (+) (negate<$>(calcula_bat_necessaria l)) y )}


verifica_viagem rota = do
    l <- apply rota
    let aux = (normalize <$> scanl (norm_sum)  batery_state l )
        return  (zipWith (+) l (calcula_bat_necessaria rota ) ,
        aux++[ normalize $(last aux ) -((last . calcula_bat_necessaria) rota )  ] )
```

A função apply é que calcula a percentagem de bateria a aumentar/diminuir em cada transição ao estado atual da bateria, para todas as possibilidades.A função verifica viagem, inicialmente aplica a função apply e posteriormente na variável aux começa a atualizar esses valores tendo em conta a bateria inicial do veículo. Por último, é retornado um par com a configuração de carga com a bateria atual em cada momento, integrando também o estado inicial e final, após o termino da viagem.

De forma a, integrar a rota neste par foi desenvolvido a seguinte função, que também filtra todos os caminhos que não são passiveis de realizar, não retornando estes casos.

```
calcula_todas_rotas node= do
    route <- hCycles node
    return $  (\x -> case ((null . snd. snd)  x) of {True-> []; False ->[x]})
              (route,(solver.result) route)
```

Após isto, temos então duas funções principais, sendo a primeira, uma função que para um dado nó, retorna os todos ciclos apartir deste, com os sítios onde pretende carregar como podemos ver no seguinte exemplo:

```
main n = (concat . calcula_todas_rotas) n

main A
-- Resultado:
[([A,B,C,D,E,F,A],([0,0,30,0,0,0],[40,38,38,18,15,13,9,5])),
([A,B,F,E,D,C,A],([0,0,70,0,0,30],[40,38,35,100,97,95,65,5])),
([A,C,D,E,B,F,A],([0,30,0,0,0,0],[40,37,17,14,13,10,6,2])),
([A,C,D,E,F,B,A],([0,30,0,0,70,0],[40,37,17,14,12,77,47,17])),
([A,F,B,E,D,C,A],([10,70,0,0,0,30],[40,44,100,96,93,91,61,1])),
([A,F,E,D,C,B,A],([0,70,0,0,0,0],[40,34,100,97,95,92,62,32]))]
```

Assim, podemos ver que para o primeiro caso é possível realizar uma viagem carregando apenas no nó C **30** % , garantindo assim o caminho ótimo, maximizando a vitalidade da bateria.

Por outro lado, para o caso da segunda main serve para calcular este output para todos os nós, podendo auxiliar o gestor do sistema a escolher a melhor rota e cidade inicial possível. Isto é conseguindo através do seguinte código:

```
main2 = (concat . calcula_todas_rotas) <$> allNodes

-- Começa em A
[[([A,B,C,D,E,F,A],([0,0,30,0,0,0],[40,38,38,18,15,13,9,5])),
([A,B,F,E,D,C,A],([0,0,70,0,0,30],[40,38,35,100,97,95,65,5])),
([A,C,D,E,B,F,A],([0,30,0,0,0,0],[40,37,17,14,13,10,6,2])),
([A,C,D,E,F,B,A],([0,30,0,0,70,0],[40,37,17,14,12,77,47,17])),
([A,F,B,E,D,C,A],([10,70,0,0,0,30],[40,44,100,96,93,91,61,1])),
([A,F,E,D,C,B,A],([0,70,0,0,0,0],[40,34,100,97,95,92,62,32]))],

-- Começa em B
[([B,A,F,E,D,C,B],([0,0,70,0,0,0],[40,10,4,71,68,66,63,60])),
([B,C,D,E,F,A,B],([0,30,0,0,0,0],[40,40,20,17,15,11,9,7])),
([B,E,D,C,A,F,B],([0,0,0,30,10,70],[40,36,33,31,1,5,70,65])),
([B,F,A,C,D,E,B],([0,0,0,30,0,0],[40,37,33,30,10,7,6,5])),
([B,F,E,D,C,A,B],([0,70,0,0,0,0],[40,37,100,97,95,35,33,31]))],

-- Começa em C
[([C,A,B,F,E,D,C],([30,0,0,70,0,0],[40,10,8,5,72,69,67,65])),
([C,A,F,B,E,D,C],([30,0,70,0,0,0],[40,10,4,69,65,62,60,58])),
([C,B,A,F,E,D,C],([0,0,0,70,0,0],[40,37,7,1,68,65,63,61])),
([C,D,E,B,F,A,C],([30,0,0,0,0,0],[40,20,17,16,13,9,6,3])),
([C,D,E,F,A,B,C],([30,0,0,0,0,0],[40,20,17,15,11,9,9,9])),
([C,D,E,F,B,A,C],([30,0,0,70,0,0],[40,20,17,15,80,50,47,44]))],

(...)
```

6 Conclusão

A realização deste projeto permitiu aplicar os conhecimentos apreendidos nas aulas teóricas e práticas da unidade curricular de Arquitetura e Cálculo, bem como incentivar a exploração de *monads* na programação funcional, de forma a, tirar máximo partido dos mesmos.

Os principais desafios durante o desenvolvimento deste projeto prenderam-se com as dificuldades a vários níveis. A primeira dificuldade foi a nível exploratório, visto que existe muita informação pouco informativa/explicativa. Outra dificuldade que tivemos foi a perceber a sintaxe monádica, como por exemplo, o caso do `do`, que aumenta a legibilidade do código, todavia, como coloca alguns operadores implicitamente (abstração para aumentar legibilidade), numa fase inicial torna a perceção do código mais complicada, pois produz-se instâncias, onde se definem certas operações que mais tarde parecem que não são utilizadas. No entanto, com o aumento da complexidade do código, é mesmo necessário a utilização dessa sintaxe(do) pois, as funções lambda vão aumentar significativamente e aí pode até tornar a função ilegível. É de realçar que, após algum treino essa dificuldade foi diminuindo.

Em suma, concluí-se que o trabalho desenvolvido se encontra bastante favorável e cumpre todos requisitos pretendidos pela equipa docente no contexto da problemática em questão e que foi desenvolvido com sucesso, uma vez que, conseguimos chegar a soluções bastante interessantes.

Referências

[1] http://www.math.uwaterloo.ca/tsp/index.html

[2] https://en.wikipedia.org/wiki/Graph_theory

[3] https://homepages.inf.ed.ac.uk/wadler/papers/marktoberdorf/baastad.pdf

[4] https://www.johndcook.com/blog/2014/03/03/monads-are-hard-because/

[5] https://medium.com/@yuriigorbylov/monads-and-why-do-they-matter-9a285862e8b4

[6] http://www.cse.chalmers.se/ rjmh/Papers/arrows.pdf