T04_G02:

Tomás Sabino Dias Costa up202205152@fe.up.pt **50%**

Manuel Mo up202205000@fe.up.pt **50%**

Tomás: rome, isStronglyConnected, shortestPath, travelSales (50%)

Manuel: cities, areAdjacent, distance, adjacent, pathDistance, travelSales (50%)

travelSales

A nossa função travelSales é em *dynamic programming*. Ou seja, o tempo de complexidade é **O(2^n)**, devido à table de *memoização* que faz com que não faça subproblemas repetidos: `type DPTable = Array (Int, Int) (Maybe Distance)`.

```
travelSales :: RoadMap -> Path
travelSales roadMap =
    let allCities = cities roadMap
        n = length allCities
        startCity = "0"  -- Começa-se pela cidade "0"
        allVisited = (1 `shiftL` n) - 1  -- Bitmask para todas as cidades
visitadas (11111101)
        dp = array ((0, 0), (n - 1, allVisited)) [((i, visited), Nothing) | i <-
[0..n-1], visited <- [0..allVisited]] -- Dynamic Programming table para guardar as
menores distâncias
    in buildPath roadMap dp startCity allVisited  -- Chama a função buildPath para
construir o caminho para o TSP
```

Na função principal, deixa-se a cidade "0" como a cidade inicial e final. De seguida, faz-se um `BitMask` para depois verificar se todas as cidades foram visitadas (exemplo: se o grafo tiver 4 cidades então o bistmask será 1111). O array `dp` é a famosa tabela de programação dinâmica que tem n linhas e 2^n colunas, onde estão apresentadas as n linhas das cidades e as*2^n colunas de todas as situações possíveis. Cria-se a tabela com tudo em Nothing. Por fim, chama a função `buildPath` com os argumentos necessários.

```
buildPath :: RoadMap -> DPTable -> City -> Int -> Path
buildPath roadMap dp startCity allVisited =
    let path = [startCity]  -- Mete a cidade inicial para a lista
        recurse currentCity visited
            | visited == allVisited = [startCity]  -- Quando todas as cidades
foram visitadas, retorna à cidade inicial
            | otherwise =
                let unvisitedCities = filter (not . testBit visited . cityIndex)
(cities roadMap) -- Seleciona as cidades ainda não visitadas
                    distances = [(distanceOrDefault roadMap currentCity nc
2147483647 -- distância da cidade atual para outra, senão existir então é INT_MAX
                                + tspDP roadMap dp startCity nc (visited `setBit`
```

```
cityIndex nc) allVisited, nc) -- Soma-se com o resultado a função tspDP e fica
(distância, nc)
                                | nc <- unvisitedCities] -- Faz-se isto para todas
as cidades não visitadas
                    nextCity = snd (minByFst distances) -- A proxima cidade será
então a cidade o menor custo
                in nextCity : recurse nextCity (visited `setBit` cityIndex
nextCity) -- Chama.se recursivamente até todas as cidades estarem visitadas e vai
adicionando-as à lista
    in path ++ recurse startCity (1 `shiftL` cityIndex startCity) -- Começa por
chamar a funçao recurse com input da primeira cidade e com a primeira cidade
visitada (ex.: 0001)
```

A função `buildPath`recebe a tabela de *memoização*, a cidade inicial e o bitmask de todas as cidades visitadas. Começa-se por armazenar na lista de retorno a primeira cidade e parte para a execução principal da função. Na função filho `recurse`, se todas as cidades forem visitadas, então é só colocar a cidade inicial no fim da lista. Se não, então cria-se uma lista de tuplos (distância, cidade), cujo primeiro elemento é o custo mínimo da cidade, visitanto todas as outras e chegando à inicial (se não existir distância de uma cidade à outra, coloca-se o maxBound, neste caso, 2147483647). De seguida, eonctra-se a cidade com o menor custo e, por fim, guardo-a e chamo novamente a função filho, porém desta vez com esta cidade visitada.

```
tspDP :: RoadMap -> DPTable -> City -> City -> Int -> Int -> Distance
tspDP roadMap dp startCity currentCity visited allVisited
    | visited == allVisited = distanceOrDefault roadMap currentCity startCity
2147483647  -- Caso base
    | otherwise =
        let cachedResult = dp ! (cityIndex currentCity, visited) `orElse`
computeAndStore
            computeAndStore =
                let result = minimum [distanceOrDefault roadMap currentCity
nextCity 2147483647 -- Distância da cidade atual para a próxima, se não exsitir
fica o INT_MAX
                                        + tspDP roadMap dp startCity nextCity
(visited `setBit` nextCityIdx) allVisited -- Soma-se a menor distância da proxima
cidade para a primeira cidade com a próxima cidade visitada
                                        | nextCity <- cities roadMap, let
nextCityIdx = cityIndex nextCity, not (testBit visited nextCityIdx)] -- Faz se
isto para todas as cidades ainda não visitadas
                in result `seq` (dp // [((cityIndex currentCity, visited), Just
result)] ! (cityIndex currentCity, visited)) `orElse` result  -- Guarda o
resultado na tabela
        in cachedResult
```

Nesta função auxiliar, ela recebe a tabela de programação dinâmica, a cidade inicial, a cidade atual, o bitmask das cidades visitadas naquele momento e o bitmask de todas as cidades visitadas. Em primeiro lugar, se todas as cidades forem visitadas, então a função retorna a distância entre a cidade atual e a cidade inicial (se não existir, retorna maxBound que é 2147483647). Caso ainda não estejam todas as cidades visitadas, a função vê se o subproblema já foi resolvido, se não resolve-se, dando o mínimo das distâncias entre a cidade atual e

todas outras cidades ainda não visitadas. De seguida, guardo o resultado para futuramente não refazer o mesmo subproblema.

shortestPath

A nossa função shortesPath é uma aproximação Brute-Force e tem complexidade exponecial devido á busca de todos os paths possíveis de um grafo. A ideia inicial foi resolver shortestPath utilizando o algoritmo de Dijkstra que tem complexidade O(E + V \log V)), mas devido a algumas dificuldades, acabamos por optar pela aproximação Brute-Force.

```
shortestPath :: RoadMap -> City -> City -> [Path]
shortestPath roadMap start end =
    let allPaths = findAllPaths roadMap start end
        pathsDistances = [(path, pathDistance roadMap path) | path <- allPaths]
        validPaths = [(path, dist) | (path, Just dist) <- pathsDistances]  --
remove possiveis Nothing Distance
        minDistance = minimum [dist | (_, dist) <- validPaths]
    in [path | (path, dist) <- validPaths, dist == minDistance]
```

A função principal simplesmente recolhe de todos os paths possíveis aqueles que têm a menor distância total.

```
findAllPaths :: RoadMap -> City -> City -> [Path]
findAllPaths roadMap start end = dfs start []
  where
    dfs current visited
        | current == end = [reverse (end : visited)]  -- reverte a ondem do
caminho criado
        | current `elem` visited = []
        | otherwise = concatMap extendPath adjacents
        where
            adjacents = adjacent2 roadMap current
            extendPath nextCity = dfs nextCity (current : visited)
```

A funcção findAllPaths recebe um grafo e duas cidades, sendo elas a cidade de partida e a cidade de chegado. A função vai fazendo dfs pelo grafo até que a cidade de chegada seja encontrada, guardando assim o caminho da cidade de partida á de chegada.

```
dfs :: RoadMap -> City -> [City] -> [City]
dfs roadMap city visited
    | city `elem` visited = [] --se estiver visitada nao adiciona
    | otherwise = city : concatMap(\adjacent -> dfs roadMap adjacent nVisited)
adjacents
    where
        adjacents = adjacent2 roadMap city
        nVisited = city : visited
```

A função dfs recebe um grafo, uma cidade atual (onde se encontra naquele momento) e uma lista de cidades já visitadas. A função percorre o grafo andando pelas cidades não visitadas, acumulando assim o caminho que vai fazendo.