Trabalho realizado por Rafael Campeão (up202207553) e Alexandre Lopes (up202207015)

Rafael Campeão (45%) funções implementadas :

- adjacent
- pathDistance
- rome
- shortestPath

Alexandre Lopes (55%) funções implementadas :

- cities
- areAdjacent
- distance
- isStronglyConnected
- travelSales

shortestPath

Para esta função foram criadas 3 novas estruturas de dados:

- AdjList - uma lista de tuplos onde o primeiro elemento de cada tuplo é uma cidade e o segundo uma lista com as cidades adjacentes e a sua distância ao primeiro elemento. Foi usada para evitar estar a recalcular adjacentes de uma cidade.
- Prelude - lista de tuplos que guarda a cada momento uma cidade e o seu adjacente mais próximo da source, no final pode ser transformada num path.
- Pqueue - lista que guarda tuplos de com uma cidade e a sua distância à source, o primeiro elemento desta lista é sempre o tuplo com a menor distância à source.

No início a função **shortestPath** inicializa uma **Pqueue** com a source e uma distância de 0, uma **lista de tuplos (City,Distance)** que mantém todas as cidades e a sua distância à

source (a distância de cada uma é inicializada a 9999 menos a source que é inicializada a 0) e uma **AdjList**. Depois chama a função auxiliar **shortestPath'** que usa o **Dijkstra's algorithm** para encontrar o caminho mais curto e calcular o seu custo. De seguida, chama a função auxiliar **allPaths**, que itera por todos os caminhos possíveis até dest, porém, se um caminho exceder o custo encontrado pelo **Dijkstra's algorithm**, esse caminho não continuará a ser explorado. Se um caminho chegar a dest e tiver um custo igual ao custo mínimo, então será adicionado à lista **paths**(argumento de allPaths). No final, essa lista que contém todos os shortestPaths encontrado, é retornada. Por último, se o **Dijkstra's algorithm** não tiver encontrado um caminho, é retornada uma lista vazia. Caso contrário, retorna a lista paths (output de allPaths)

shortestPath' - Recebe como argumentos um Roadmap, uma source(cidade), uma dest(cidade), uma disToSource, uma Pqueue, uma AdjList e um Prelude e no fim retorna um Prelude. Pega no primeiro elemento da Pqueue que recebe e encontra os seus adjacentes. Usa a função getNewDist para verificar se é possível baixar a distância de alguma cidade à source, e, de acordo com esse resultado, dá update à priority queue, à distToSource e ao prelude. Faz uma chamada recursiva até receber uma priority queue vazia.

allPaths - Recebe como argumentos um Roadmap, uma AdjList, uma Distance (custo mínimo para os caminhos), uma source(cidade que está a ser verificada), uma dest(cidade final), uma lista de cidades(cidades já visitadas), uma lista de paths (lista que vai ser retornada no final), um Path(caminho que está a ser explorado) e uma Distance (custo do caminho a ser explorado). Se source e dest forem iguais, a distância do path é calculada. Se esta for igual ao custo mínimo, o path é adicionado à lista de paths; caso contrário é descartado. Se source e dest forem diferentes, é chamada uma função auxiliar para todos os adjacentes da source. Se o adjacente já tiver sido explorado ou se o custo do caminho mais a distância do adjacente à source for maior que o custo mínimo, o path é descartado. Caso contrário, allPaths é chamada recursivamente, atualizando a source, a lista de cidades visited, o path que está a ser explorado e o seu custo.

travelSales

A função **travelSales** resolve o *Travel Salesman Problem* (TSP), ou seja, encontra a rota mais curta que passa por todas as cidades exatamente uma vez e retorna à cidade original. Para esta função, foi usada uma abordagem de programação dinâmica, utilizando uma tabela de memoização e *bitmasking*, o que ajuda a evitar cálculos redundantes.

A tabela de memoização é implementada como um array de arrays, onde cada entrada corresponde a um par de *bitmask*, que representa as cidades visitadas, e a cidade atual. Isto permite armazenar e reutilizar os resultados de subproblemas, evitando repetir o cálculo de distâncias nas chamadas recursivas.

O algoritmo representa o estado atual com um *bitmask*, onde cada bit indica se uma cidade foi visitada. Por exemplo, se tivermos quatro cidades (0, 1, 2 e 3), um *bitmask* de 1010 indica que as cidades 1 e 3 foram visitadas.

A função **tsp** é definida recursivamente e recebe como parâmetros o *bitmask*, a cidade atual e a tabela de memoização. A invocação inicial de **tsp** é feita na função **travelSales**, começando com o *bitmask* que representa que a cidade inicial foi visitada (1) e a posição inicial (0).

O **tsp** começa por verificar se todas as cidades foram visitadas, comparando o *bitmask* com 2^n-1, onde n é o número total de cidades. Quando todas as cidades tiverem sido visitadas, a função calcula a distância de retorno à cidade inicial através da função **distance**, que retorna a distância entre duas cidades.

Enquanto houver cidades por visitar, a função faz chamadas recursivas para **tsp** para cada uma das cidades não visitadas, atualizando o *bitmask* de forma a incluir a nova cidade visitada. Durante este processo, o algoritmo também calcula a distância acumulada até à cidade atual e a distância da cidade atual até à nova cidade que está a ser visitada.

Os resultados das chamadas recursivas são armazenados numa lista chamada possiblePaths, que contém as distâncias acumuladas e os caminhos correspondentes. Assim, o algoritmo procura nessa lista o caminho de menor custo.

Depois de determinar o caminho de menor custo para o conjunto atual de cidades visitadas, a tabela de memoização é atualizada com o resultado encontrado.

A função **tsp** retorna um tuplo que contém o menor caminho e a tabela de memoização atualizada.

Por fim, na função **travelSales**, o resultado da invocação de **tsp** é obtido através de uma verificação:

- Se o **tsp** retornou Nothing, não existem caminhos válidos pelo que a função retorna uma lista vazia.
- Se retornar um caminho válido, este será extraído do tuplo e retornado pela função **travelSales**.