I. INTRODUÇÃO

Numa situação real, os problemas que se procuram resolver, muitas vezes não oferecerem as caraterísticas ideais para aplicar os algoritmos habituais de determinação de melhores caminhos. Com este projeto tem-se a oportunidade de analisar um problema com uma estrutura hierárquica e que contém limitações a nível económico. Neste sentido, é necessário conciliar a procura do caminho de menor custo com o conjunto de regras de encaminhamento que o problema exige.

II. NOTAS ACERCA DOS ALGORITMOS

Na execução do programa optou-se pelo armazenamento da informação da rede num vetor de listas de adjacências, em que o índice do vetor corresponde à identificação numérica (ID) do ISP. Cada elemento do vetor aponta para uma lista ligada que contém a informação dos seus vértices vizinhos, inclusive o tipo de rota que utiliza para chegar a estes. À partida, o programa não sabe o número de vértices na rede (nem o seu ID), o que implica a inicialização do vetor com um tamanho grande o suficiente para garantir que todos os IDs tenham um índice correspondente. Este processo, caso os vértices não sejam numerados sequencialmente, produz índices no vetor que não se encontram ligados a mais nenhum vértice. Contudo, o programa está preparado para essas ocorrências. Durante a execução dos algoritmos, assume-se que os vértices sem qualquer vizinho não fazem parte da rede e são, por isso, ignorados. Do ponto de vista de gestão de memória não é a solução mais eficiente, contudo esse não é o fator mais relevante na resolução deste problema. A grande vantagem é o acesso às posições dos vértices ser O(1).

Por pré-definição o programa lê a rede do ficheiro "LargeNetwork.txt", contudo poderá ser utilizado qualquer ficheiro de texto se for dado como argumento na execução da aplicação.

III. ALGORITMOS UTILIZADOS

A. Search for customer cycles:

O algoritmo que se procura desenvolver é baseado no problema de encontrar ciclos num grafo direcionado. Neste tipo de problemas é usado um algoritmo de travessia, como o DFS (*Depth-First Search*), para gerar uma árvore caso o grafo seja conexo. Se durante o processo de gerar a árvore for encontrada uma *back-edge* (uma aresta que une um certo vértice a si próprio ou a um dos seus antecessores na árvore), então é possível concluir que existem ciclos no grafo em questão.

Numa rede que se rege pelas regras de Inter-AS Routing pode-se aplicar um método muito idêntico ao utilizado para encontrar ciclos num grafo normal. A grande diferença é que para este caso, em concreto, são ignoradas todas as ligações que não sejam do tipo *customer*, já que só existem customer cycles se, partindo de um vértice for possível voltar a ele apenas por rotas do tipo *customer*.

Para detetar uma back-edge, vai-se guardando a informação sobre os vértices que foram visitados durante a travessia numa *Stack*. O processo da travessia pelo grafo e da inserção na *Stack* é feito recursivamente. Se for alcançado um vértice que já estava na *Stack* (uma back-edge), então existe um ciclo.

Usa-se um vetor *visited* que marca todos os vértices visitados pelo algoritmo, e um vetor *stack* que marca todos os vértices marcados durante a travessia do ramo atual. Estes dois vetores são necessários já que se pode chegar duas vezes ao mesmo vértice já visitado mas por caminhos diferentes. Como o grafo é direcionado e condicionado, isso não constitui um ciclo, daí o uso da *stack* para manter apenas a informação do caminho.

Algorithm 1 checkCycle(graph)

```
1: //initialization
2: for each v do
3:     visited[v] = FALSE;
4:     stack[v] = FALSE;
5: for each v do
6:     if checkCycleRec(v, visited, stack, graph) then
7:         return TRUE //there's a cycle
8: return FALSE //there are no cycles
```

Após a execução deste algoritmo, chega-se a dois resultados possíveis: o grafo ter customer cycles ou o grafo não ter customer cycles. Caso se conclua que o grafo não tem customer cycles, então pode-se prosseguir com o resto do programa, executando os restantes algoritmos. Por outro lado, se o resultado for a existência de customer cycles, então é necessário abortar o programa, uma vez que não há garantias que a procura por melhores caminhos chegue a um estado estável. A instabilidade deriva de, neste ciclo, os vértices atribuírem para melhor caminho a rota que lhes garanta menor custo (rota de cliente), o que pode resultar em que o melhor caminho gerado passe pelo vértice inicial e que por isso induza um loop continuo que não permita percorrer outros vértices.

Algorithm 2 checkCycleRec(v, visited, stack, graph)

```
1: if v not yet visited then
2:     visited[v] = TRUE;
3:     stack[v] = TRUE;
4:     while there are adjacent nodes to v do
5:         if adjacent node w is a customer of v then
6:             if w not visited and checkCycleRec(w, visited, stack, graph) then
7:                 return TRUE //there's a cycle
8:             else if w is in stack then
9:                 return TRUE //there's a cycle
10: //there is no cycle in this branch
11: Stack[v] = FALSE //after fully explored remove v from stack when recurring
12: return FALSE
```

A complexidade deste algoritmo é a mesma que a DFS para uma lista de adjacências, O(V+E), já que no pior caso (não haver ciclo) visitam-se todos os vértices e arestas uma única vez.

B. Checking if the network is commercialy connected:

Sendo uma rede comercialmente conexa, uma rede com a propriedade de que todos os ISPs conseguem alcançar qualquer outro ISP da rede, partiu-se do principio que se poderia utilizar os mesmos algoritmos que são usados para testar se um grafo é fortemente conexo.

Existem muitos algoritmos para resolver este tipo de problema. Uma ideia simples seria efetuar V vezes uma DFS começando em cada vértice. Caso uma das DFS não visitasse todos os vértices, então o grafo não seria fortemente conexo. Este algoritmo teria uma complexidade igual a O(V*(V+E)). Apesar deste procedimento, não ter sido utilizado para resolver o problema em questão, as implicações deste algoritmo ajudaram na intuição do algoritmo para descobrir o tipo de rotas que se discutirá na secção seguinte.

Numa tentativa de conseguir uma complexidade inferior, tentou-se aplicar o algoritmo de Strongly Connected Components, também conhecido como Kosaraju's algorithm. A ideia chave deste algoritmo é que se de um dado vértice se conseguir alcançar todos vértices e se de todos os vértices for possível alcançar esse mesmo vértice, então pode-se concluir que o grafo é fortemente conexo. A complexidade deste algoritmo é idêntico ao de uma DFS aplicada duas vezes, o que equivale a O(V+E), de modo que teria um desempenho muito apelativo.

Contudo, apesar de, intuitivamente, este algoritmo parecer funcionar para descobrir se a rede é comercialmente conexa, verificou-se, que dadas as condições deste problema (haver rotas inválidas), o algoritmo de Kosaraju não funciona.

Realizando uma análise mais profunda da rede e das suas caraterísticas, observou-se que os vértices Tier1 (no topo da hierarquia) têm um papel fundamental na conexão do grafo. Caso não haja customer cycles, pode ser provado pelos seguintes factos derivados das condições de exportações de rotas e sentido *cashflow*:

- Todos os *providers* têm uma rota que chega a todos os seus clientes;
- Todos os nós têm obrigatoriamente uma rota de *providers* até um Tier1;
- Dada a direção do cash flow, qualquer vértice tem uma rota possível para todos os vértices que partilhem o seu *provider*;
- Todos os nós têm uma rota para os clientes dos seus *peers*
- Numa ligação *peer* entre providers, todos os clientes dos dois *provideres* têm rota válida entre si;
- Então, qualquer vértice *x* tem rota para todos os vértices que partilhem o seu *provider* e para vértices cujo *provider* tenha relação *peer* com o *provider* do vértice *x*;
- Como todos os vértices têm rota para pelo menos um tier1, se todos os tier1 tiverem ligação *peer* entre si, então qualquer vértice tem uma rota válida para todos os vértices da rede (condição necessária e suficiente).

O primeiro passo do algoritmo é descobrir quais são os vértices do grafo que correspondem a ISPs Tier1. Esta etapa não é efetuada na função abaixo descrita, mas sim durante a própria leitura do ficheiro e criação do grafo. Em primeiro lugar, inicializa-se a 1 uma Flag presente em cada vértice que indica se um dado ISP é tier1. Durante a análise de cada linha do ficheiro de texto, sempre que for introduzida uma route *provider* de um certo vértice origem para um vértice destino, altera-se a Flag de Tier1 do vértice para 0. No fim da leitura do ficheiro, todos os vértices que ainda mantenham a Flag a 1 serão Tier1. Este método permite evitar percorrer um algoritmo de travessia para encontrar estes ISPs e poupando, desta forma, tempo de execução.

Algorithm 3 commercialy_connected(network)

```
1: for each Node do
2:     if Node == Tier1 and Node has neighbours then
3:         if checkTier1Connections(List, network, Node) == TRUE then
4:             continue //keeps searching for the next Tier1
5:         else
6:             return FALSE // it isn't commercialy connected
7: return TRUE // it is commercialy connected
```

O algoritmo consiste em percorrer a lista de adjacências à procura dos vértices com a Flag de Tier1 ativa. Para apoiar a execução do algoritmo, é utilizada uma lista onde são guardados os vértices Tier1 à medida que são encontrados. Quando é encontrado um Tier1, antes de ser inserido na lista, é testado se tem uma rota *peer* para todos os Tier1 que já se encontram na lista. Caso tenha, continua-se a percorrer a lista de adjacências à procura do próximo Tier1, caso contrário conclui-se que a rede não é comercialmente conexa.

A complexidade dominante corresponde à etapa de testar se um Tier1 está ligado a todos os outros. A adição e comparação sucessiva de vértices, significa que a cada iteração ter-se-á que efetuar N+1 comparações, o que corresponde a uma complexidade temporal de $O(\frac{N(N+1)}{2}) \approx O(N^2)$, sendo N o número de vértices do tipo Tier1.

Caso a rede seja constituída apenas por ISPs do tipo Tier1 todos ligados entre si por rotas *peer*, implicará que o número de operações seja da ordem $O(N^2)$. No entanto, como numa rede de ISPs real o número de vértices Tier1 é muito pequeno face ao número total de vértices da rede, então a complexidade temporal quadrática não é um fator muito preocupante no desempenho o algoritmo.

Algorithm 4 checkTier1Connections(List, network, node_S)

```
1: //with node_S being a tier1 node
2: if List is empty then
3:     insert node_S in the list
4:     return TRUE
5: while end of list hasnt been reached do
6:     while node_S adjacency list end hasnt been reached do
7:         if tier1 in the list is Peer of node_S then
8:             break // move to next Tier1 in the list
9:         next neighbour of node_S
10:    if node_S adjacency list was searched until the end then
11:        return FALSE // node_S isn't connected to this Tier1
12:    next Tier1 of the list
13: Insert node_S at the head of the list
14:
15: return TRUE //node_S is connected to all tier1 nodes analysed so far
```

C. Checking the type of route elected at each node to reach the destination:

A abordagem mais intuitiva a este problema seria percorrer todos os vértices e para cada um encontrar o tipo de rota para o vértice destino. Obviamente esta solução é dispendiosa do ponto de vista temporal, uma vez que executa várias operações desnecessárias, como analisar caminhos para vértices irrelevantes.

Para obter uma solução mais eficiente pode-se adaptar um algoritmo de caminhos curtos, como o algoritmo de Dijsktra com recurso a um acervo, às condições do problema. Considerando-se o vértice destino como o vértice de onde começa o algoritmo e o relaxamento feito olhando para as ligações no sentido contrário, é possível calcular o tipo de rotas de todos os vértices para um destino.

É retirado sempre do topo do acervo o vértice que corresponde à melhor rota disponível naquela iteração, usando a ordem de preferência (total order) $1 \succeq 2 \succeq 3 \succeq 4$, que correspondem às rotas do tipo *customer*, *peer*, *provider*, *non existent*, respetivamente. A operação seletiva para o problema em questão é a minimização do custo comercial e o relaxamento é a atualização conforme se encontra rotas de menor custo, que coincide com o algoritmo de Dijkstra. A validação do método pode ser feita através da figura (2) que representa o desempenho do algoritmo para a rede da figura (1).

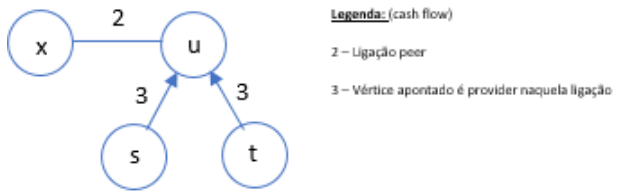

Fig. 1. Exemplo de Rede

Para o exemplo da figura 1, para saber o tipo de rotas para o destino *s*, o algoritmo tomará esse vértice como origem. O procedimento está representado na tabela da figura 2, em que a amarelo estão marcados os vértices que são extraídos do acervo:

Extracted	s	u	t	x	loop
-	0	4	4	4	init
s	0	1	4	4	1
s, u	0	1	3	2	2
s, u, x	0	1	3	2	3
s, u, x, t	0	1	3	2	4

Fig. 2. Algoritmo aplicado à Rede da figura 1

Podemos explicar a execução do algoritmo aplicado tomando como exemplo a rede da figura (1), em que o destino é o vértice *s*:

- O vértice destino *s* é inicializado com prioridade 0 para ser extraído imediatamente do acervo;
- Depois de extraído o vértice *s*, na iteração 1, é descoberto o vértice *u*, que é *provider* de *s*. Logo, *u* terá uma rota de *customer* (tipo 1) para o destino.
- Como o tipo de rota que *u* tinha até aquele momento era *unreachable* (tipo 4), a rota recém-descoberta é melhor ($1 \succeq 4$) e, por isso, o vetor com os tipos de rota é atualizado para o vértice *u*.
- Um vértice extraído não volta a ser revisitado (greedy algorithm);
- Na iteração seguinte, extrai-se o vértice com maior prioridade, ou seja, com a rota de menor custo. Neste caso, será o vértice *u*.
- O vértice *u* descobre que tem dois vizinhos *x* e *t*. Devido às regras do protocolo BGP, é necessário averiguar se a rota que o seu vizinho utiliza para chegar a si é compatível com a rota que *u* usa para chegar ao destino.
- Como tanto uma rota *peer* seguida de uma rota *customer*, como uma rota *provider* seguida de uma rota *customer* são caminhos válidos, então o vetor dos tipos de rotas é atualizado para os dois vértices.
- O processo repete-se até terem sido percorridos todos os vértices da rede ou, caso seja comercialmente conexo, se extrair do acervo, uma rota que corresponda ao tipo *provider*.

Saber que uma rede é comercialmente conexa permite retirar uma conclusão muito útil quando se está a realizar este algoritmo. Pelo algoritmo de cálculo de rotas, as rotas que são analisadas em primeiro lugar serão sempre aquelas que oferecem menor custo, isto é, as rotas de *customer* e de *peer*. Como o atributo de comercialmente conexo garante que o grafo não é desconexo, se for retirada uma rota *provider* então, garantidamente, todas as outras rotas que resta analisar serão do tipo *provider*. Desta forma, podemos atualizar o número de rotas *provider* da rede e parar o algoritmo.

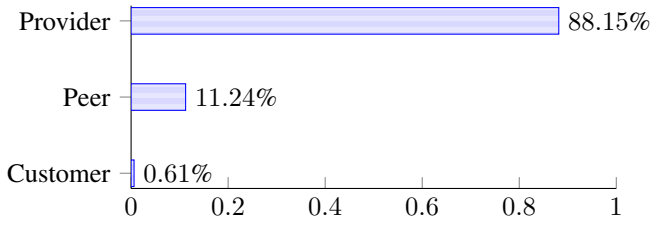

Fig. 3. Histogram for LargeNetwork

No histograma da figura (3) encontram-se as estatísticas acerca dos tipo de rotas tomado por todos os vértices para todos os destinos da rede LargeNetwork.txt, uma rede comercialmente conexa com aproximadamente 44000 vértices. Pode-se observar que a esmagadora maioria das rotas tomadas são do tipo *provider*, logo a paragem do algoritmo ao atingir uma rota *provider* melhora drasticamente a eficiência do programa.

Caso a rede não seja comercialmente conexa, não é possível interromper o algoritmo ao extrair uma rota do tipo *provider*, dado que ainda podem existir vértices no acervo que ainda não foram visitados.

Algorithm 5 electedRoute(network, dest, n_Proutes, n_Rroutes, n_Croutes, commercialFlag, heap)

```
1:
2: // initialization of heap and type array
3: for all v nodes do
4:       type[v] = UNREACHABLE //vector that has the type of route for each node
5:       if v node doesnt have neighbours then
6:             if node is the destination then
7:                   type[v] = 0 //so this node is the first extracted from the heap
8:             initialize_heap()
9:  updateHeap(heap, dest, type[dest]) //update destin's priority and heapify
10:
11: while !isEmpty(heap) do
12:       minHeapNode = extractMin(heap) // heapify after extraction
13:       if network is Commercially Connected then
14:             if extracted node has provider route then
15:                   updateStatistics();
16:                   return //stops the algorithm
17:       updateStatistics()
18:       while adjacency_node != NULL do
19:             id = adjacency_node->id
20:             // looks at the link in opposite direction
21:             invType = invertroute(adjacency_node->type)
22:             if nodeInTheHeap() and type[id] != unreachable then
23:                   if newRoute < prevRoute and routeIsValid() then
24:                         type[id] = invType //update routes vector
25:                         updateHeap(heap, id, type[id])
26:             next adjacency_node
27: return
```

A complexidade temporal deste algoritmo provém de dois grandes blocos: O processo de visitar e explorar todos os vértice O(V+E) e o processo de, para cada vértice, ajustar a sua posição em função da sua prioridade (Heapify), usando uma heap binária, que tem complexidade O(logV). Logo a complexidade temporal total é O((V+E)*logV) $\approx$ O(ElogV), em que V é o número de vértices e E o número de arestas.

IV. Considerações Finais:

Como foi estudado no inicio do projeto, os customer cycles neste tipo de rede podem originar problemas na deteção de caminhos ideais. Uma solução para prevenir que estes ciclos resultem na instabilidade do programa poderia ser impedir que os algoritmos de procura em profundidade atravessem duas vezes o mesmo vértice numa travessia. No entanto, se este problema de encaminhamento tivesse que ser pensado de forma distribuída, então a abordagem teria que ser diferente. Por exemplo, o ISP teria que anunciar de alguma forma aos seus vizinhos que uma certa mensagem já passou por si.

Apesar do algoritmo utilizado para encontrar os tipos de rota ter sido otimizado face ao problema deste projeto e de conseguir um desempenho bastante superior ao de algoritmos mais intuitivos, existem maneiras de diminuir a complexidade deste.

Uma forma de melhorar o desempenho do algoritmo de eleição das rotas para um destino, seria utilizar outro tipo de estrutura de dados como auxiliar. Uma possibilidade seria utilizar três filas, uma para cada tipo de rota. Utilizando o sistema *First-In First-Out*, adicionava-se as rotas às filas há medida que iam sendo descobertas e, desta forma, seriam guardados os próximos vértices a analisar para cada tipo de ligação. Extrai-se os vértices de cada fila até esta ficar vazia, começando pela fila dos *customers*, depois dos *peers* e por fim *providers*. Utilizando este método não seria necessário ordenar nenhum vetor, como é feito no caso do acervo, por isso conseguir-se-ia complexidades temporais de extração dos vértices de O(1). Seria possível, por este meio, em troca de um maior consumo de memória, alcançar uma complexidade temporal para a procura dos tipos de rota na ordem de O(V), sendo V o número de vértices do grafo.