Modelação e Desempenho de Redes e Serviços

Mini-Project - Nº 2

João Monteiro (102690), João Gaspar (107708)

Departamento de Eletrónica, Telecomunicações e Informática (DETI)

Universidade de Aveiro

19 de dezembro de 2024

Conteúdo

Lista de Figuras

Introdução

O presente trabalho, intitulado *Traffic Engineering of Telecommunication Networks*, aborda a problemática da engenharia de tráfego em redes de telecomunicações, com um foco específico em redes baseadas no protocolo *Multiprotocol Label Switching* (MPLS). A rede em análise pertence a um *Internet Service Provider* (ISP) e apresenta uma topologia composta por 16 nós e 28 ligações, distribuídas numa área retangular de 1200 km por 600 km. Todas as ligações possuem uma capacidade simétrica de 100 Gbps, estando os seus comprimentos especificados na matriz quadrada L.

A rede suporta três tipos de serviços: dois serviços *unicast*, designados por $s = 1$ e $s = 2$, nos quais o tráfego é enviado entre um nó de origem e um nó de destino específicos, e um serviço *anycast*, designado por $s = 3$, no qual o tráfego é encaminhado para o nó mais próximo, tendo em consideração o menor atraso de propagação entre os possíveis destinos. Os parâmetros dos fluxos de tráfego são definidos pela matriz T, que especifica, para cada fluxo, o tipo de serviço, os nós de origem e destino, e os débitos de tráfego em ambas as direções. Para calcular o atraso de propagação, considera-se a velocidade da luz nas fibras óticas, fixada em $v = 2 \times 10^5$ km/s, sendo este valor utilizado para derivar a matriz D, que contém os atrasos de propagação em cada direção de cada ligação da rede.

O objetivo deste estudo é explorar e aplicar conceitos avançados de engenharia de tráfego e otimização em redes de telecomunicações, analisando métricas de desempenho como o atraso médio, o pior atraso de ida e volta (*round-trip delay*) e a carga máxima dos enlaces da rede. Pretende-se, ainda, identificar soluções que maximizem a eficiência e a robustez da rede em cenários realistas, proporcionando um melhor entendimento teórico e prático dos desafios inerentes à engenharia de tráfego e à gestão de redes de alta capacidade.

Capítulo 1

Task 1

1.1 Exercício 1.a.

1.1.1 Código

```
%% 1.a.

clear
clc

fprintf('------------------------------ Task 1.a
    .------------------------------\n');

% carregar os dados
load('InputDataProject2.mat');

% par metros
nNodes = size(Nodes, 1);
nFlows = size(T, 1);
nLinks = size(Links, 1);

v = 2 * 10^5;
D = L / v;

anycastNodes = [3 10];

% inicializar vari veis para os atrasos e caminhos
Taux = zeros(nFlows, 4);
delays = zeros(nFlows, 1);
sP = cell(nFlows, 1);
nSP = cell(nFlows, 1);

% calcular os caminhos mais curtos e os atrasos de ida e volta
for n = 1:nFlows
    if T(n, 1) == 1 || T(n, 1) == 2
        [shortestPath, totalCost] = kShortestPath(D, T(n, 2), T(n, 3),
            1);
        sP{n} = shortestPath;
        nSP{n} = length(shortestPath);
        delays(n) = totalCost;
        Taux(n, :) = T(n, 2:5);
    elseif T(n, 1) == 3
        if ismember(T(n, 2), anycastNodes)
```

```
37              sP{n} = {T(n, 2)};
38              nSP{n} = 1;
39              Taux(n, :) = T(n, 2:5);
40              Taux(n, 3) = T(n, 2);
41          else
42              cost = inf;
43              Taux(n, :) = T(n, 2:5);
44              for i = anycastNodes
45                  [shortestPath, totalCost] = kShortestPath(D, T(n, 2),
                        i, 1);
46                  if totalCost < cost
47                      sP{n} = shortestPath;
48                      nSP{n} = 1;
49                      cost = totalCost;
50                      delays(n) = totalCost;
51                      Taux(n, 3) = i;
52                  end
53              end
54          end
55      end
56  end
57
58  unicastFlows1 = find(T(:, 1) == 1);
59  unicastFlows2 = find(T(:, 1) == 2);
60  anycastFlows = find(T(:, 1) == 3);
61
62  maxDelayUnicast1 = max(delays(unicastFlows1)) * 2 * 1000;
63  avgDelayUnicast1 = mean(delays(unicastFlows1)) * 2 * 1000;
64
65  maxDelayUnicast2 = max(delays(unicastFlows2)) * 2 * 1000;
66  avgDelayUnicast2 = mean(delays(unicastFlows2)) * 2 * 1000;
67
68  maxDelayAnycast = max(delays(anycastFlows)) * 2 * 1000;
69  avgDelayAnycast = mean(delays(anycastFlows)) * 2 * 1000;
70
71  % mostrar os resultados
72  fprintf('Anycast nodes: %d  %d\n', anycastNodes(1), anycastNodes(2));
73  fprintf('Worst round-trip delay (unicast service 1): %.2f ms\n',
        maxDelayUnicast1);
74  fprintf('Average round-trip delay (unicast service 1): %.2f ms\n',
        avgDelayUnicast1);
75  fprintf('Worst round-trip delay (unicast service 2): %.2f ms\n',
        maxDelayUnicast2);
76  fprintf('Average round-trip delay (unicast service 2): %.2f ms\n',
        avgDelayUnicast2);
77  fprintf('Worst round-trip delay (anycast service): %.2f ms\n',
        maxDelayAnycast);
78  fprintf('Average round-trip delay (anycast service): %.2f ms\n',
        avgDelayAnycast);
```

Este código foi adaptado dos exercícios da *Task12* dos guiões das aulas, com o objetivo de calcular os atrasos de ida e volta (*round-trip delays*) para os serviços *unicast* e *anycast* suportados pela rede. A matriz de atrasos de propagação (D) foi obtida com base na velocidade da luz nas fibras óticas e utilizada para determinar os caminhos mais curtos.

Os fluxos *unicast* ($s = 1$ e $s = 2$) foram roteados pelos caminhos de menor atraso entre os nós de origem e destino. Já os fluxos *anycast* ($s = 3$) foram encaminhados para o nó *anycast* mais próximo, avaliando-se os atrasos

até todos os nós disponíveis.

Por fim, o código calcula e apresenta o pior atraso e o atraso médio para cada serviço, permitindo a análise detalhada do desempenho da rede. A abordagem reflete uma aplicação prática de conceitos fundamentais de engenharia de tráfego e otimização de redes.

1.1.2 Resultados e Conclusões

No *exercício 1.a.*, foram analisados os atrasos de ida e volta (*round-trip delays*) para os três tipos de serviço suportados pela rede. Os nós *anycast* foram definidos como os nós 3 e 10, e os resultados obtidos são os da figura abaixo (Figura 1.1).

```
------------------------------ Task 1.a.------------------------------
Anycast nodes: 3  10
Worst round-trip delay (unicast service 1): 9.04 ms
Average round-trip delay (unicast service 1): 5.42 ms
Worst round-trip delay (unicast service 2): 11.07 ms
Average round-trip delay (unicast service 2): 5.83 ms
Worst round-trip delay (anycast service): 6.16 ms
Average round-trip delay (anycast service): 3.43 ms
```

Figura 1.1: Resultados correspondentes ao exercício 1.a.

A análise dos resultados permite tirar as seguintes conclusões:

1. O **serviço *unicast* 1** apresenta atrasos menores, tanto no pior cenário quanto na média, em comparação com o serviço *unicast* 2. Isto deve-se à distribuição do tráfego e ao número de fluxos associados a cada serviço.

2. O **serviço *anycast*** apresenta os menores valores de atraso, tanto médio quanto no pior caso. Este comportamento reflete a escolha do nó de destino mais próximo, o que minimiza os atrasos totais de propagação.

A diferença significativa entre os serviços *anycast* e *unicast* é explicada pela arquitetura do *anycast*, que permite uma maior flexibilidade na escolha do nó de destino. Enquanto os fluxos *unicast* estão restritos a um nó de destino predefinido, os fluxos *anycast* podem ser roteados para o nó mais próximo, com base no menor atraso de propagação. Este mecanismo reduz os atrasos ao explorar eficientemente a topologia da rede.

Este resultado é consistente com as expectativas teóricas, uma vez que o *anycast* foi projetado para otimizar a entrega de tráfego em cenários de múltiplas opções de destino. Na prática, isso significa que o *anycast* tende a apresentar menores atrasos, especialmente em redes densamente conectadas,

como a analisada neste exercício. Em contrapartida, os fluxos *unicast* não possuem essa flexibilidade, resultando em atrasos maiores, dependendo da localização dos nós de origem e destino e da congestionamento das rotas.

1.2 Exercício 1.b.

1.2.1 Código

```
%% 1.b.

clear
clc

fprintf('------------------------------ Task 1.b
    .------------------------------\n');

% carregar os dados
load('InputDataProject2.mat');

% par metros
nNodes = size(Nodes, 1);
nFlows = size(T, 1);
nLinks = size(Links, 1);

v = 2 * 10^5;
D = L / v;

anycastNodes = [3 10];

% inicializar vari veis para os atrasos e caminhos
Taux = zeros(nFlows, 4);
delays = zeros(nFlows, 1);

% calcular os caminhos mais curtos e os atrasos de ida e volta
for n = 1:nFlows
    if T(n, 1) == 1 || T(n, 1) == 2
        [shortestPath, totalCost] = kShortestPath(D, T(n, 2), T(n, 3),
            1);
        sP{n} = shortestPath;
        nSP{n} = length(shortestPath);
        delays(n) = totalCost;
        Taux(n, :) = T(n, 2:5);
    elseif T(n, 1) == 3
        if ismember(T(n, 2), anycastNodes)
            sP{n} = {T(n, 2)};
            nSP{n} = 1;
            Taux(n, :) = T(n, 2:5);
            Taux(n, 3) = T(n, 2);
        else
            cost = inf;
            Taux(n, :) = T(n, 2:5);
            for i = anycastNodes
                [shortestPath, totalCost] = kShortestPath(D, T(n, 2),
                    i, 1);
                if totalCost < cost
                    sP{n} = shortestPath;
                    nSP{n} = 1;
                    cost = totalCost;
                    delays(n) = totalCost;
```

```
                    Taux(n, 3) = i;
                end
            end
        end
    end
end

% calcular as cargas dos links
Loads = calculateLinkLoads(nNodes, Links, Taux, sP, ones(nFlows, 1));

% encontrar a pior carga de link
worstLinkLoad = max(max(Loads(:, 3:4)));

% mostrar os resultados
fprintf('Anycast nodes = %d  %d\n', anycastNodes(1), anycastNodes(2));
fprintf('Worst link load = %.2f Gbps\n', worstLinkLoad);
for i = 1:nLinks
    fprintf('{%d-%d}: %.2f %.2f\n', Loads(i, 1), Loads(i, 2), Loads(i,
        3), Loads(i, 4));
end
```

Este código foi adaptado dos exercícios da *Task 12* dos guiões das aulas, com o objetivo de calcular as cargas de todos os links da rede e identificar a pior carga (*worst link load*). A matriz de atrasos de propagação (D) foi utilizada para determinar os caminhos mais curtos para os fluxos *unicast* e *anycast*.

Os fluxos *unicast* ($s = 1$ e $s = 2$) foram roteados pelos caminhos de menor atraso entre os nós de origem e destino, enquanto os fluxos *anycast* ($s = 3$) foram encaminhados para o nó *anycast* mais próximo. As cargas dos links foram calculadas com base nas rotas resultantes, permitindo identificar o link mais utilizado.

O código permite uma análise detalhada do desempenho da rede, demonstrando a aplicação prática de conceitos de engenharia de tráfego e otimização de redes.

1.2.2 Resultados e Conclusões

No *exercício 1.b.*, foi analisada a carga de todos os links da rede e identificada a pior carga de link (*worst link load*). Os nós *anycast* foram definidos como os nós 3 e 10, e os resultados obtidos encontram-se na figura abaixo (Figura 1.2).

A análise dos resultados permite tirar as seguintes conclusões:

1. A pior carga de link (**98.20 Gbps**) ocorre no link **9-10**, o que era esperado devido ao fluxo significativo de tráfego na região central da rede.

2. Os links que conectam nós periféricos apresentam cargas menores ou nulas, como os links *1-7*, *13-16* e *15-16*. Isso reflete o baixo tráfego direcionado a essas áreas.

```
------------------------------ Task 1.b.------------------------------
Anycast nodes = 3  10
Worst link load = 98.20 Gbps
{1-2}: 15.00 15.20
{1-5}: 28.70 26.20
{1-7}: 0.00 0.00
{2-3}: 50.40 52.00
{2-4}: 33.10 34.20
{2-5}: 47.80 48.90
{3-6}: 31.50 9.00
{3-8}: 33.10 31.60
{4-5}: 36.00 38.70
{4-8}: 34.40 35.20
{4-9}: 13.40 15.60
{4-10}: 46.00 46.90
{5-7}: 47.80 48.90
{6-8}: 11.60 9.90
{6-14}: 38.90 29.40
{6-15}: 8.50 3.00
{7-9}: 64.40 71.10
{8-10}: 85.20 88.00
{8-12}: 14.80 14.10
{9-10}: 87.90 98.20
{10-11}: 85.60 82.90
{11-13}: 61.70 55.60
{12-13}: 15.00 17.10
{12-14}: 14.80 14.10
{13-14}: 40.90 40.80
{13-16}: 0.00 0.00
{14-15}: 29.30 29.40
{15-16}: 0.00 0.00
```

Figura 1.2: Resultados correspondentes ao exercício 1.b.

3. Os fluxos *anycast* ajudam a distribuir melhor o tráfego entre os nós *3* e *10*, reduzindo os atrasos e otimizando a utilização da rede.

Em geral, os resultados obtidos são coerentes com o comportamento esperado para redes que utilizam serviços *unicast* e *anycast*. A concentração de tráfego em links centrais ressalta a importância de estratégias de engenharia de tráfego para mitigar congestionamentos e otimizar o desempenho da rede.

1.3 Exercício 1.c.

1.3.1 Código

```
1  %% 1.c.
2
3  clear
4  clc
5
6  fprintf('------------------------------ Task 1.c
       .------------------------------\n');
7
8  % carregar os dados
9  load('InputDataProject2.mat');
```

```
10
11  % par metros
12  nNodes = size(Nodes, 1);
13  nFlows = size(T, 1);
14  nLinks = size(Links, 1);
15
16  v = 2 * 10^5;
17  D = L / v;
18
19  % inicializar vari veis para os melhores resultados
20  bestAnycastNodes = [];
21  minWorstLinkLoad = inf;
22  bestDelays = [];
23  bestTaux = [];
24  bestSP = [];
25
26  % testar todas as combina  es poss veis de dois n  s
27  for i = 1:nNodes
28      for j = i+1:nNodes
29          anycastNodes = [i j];
30
31          % inicializar vari veis
32          Taux = zeros(nFlows, 4);
33          delays = zeros(nFlows, 1);
34          sP = cell(nFlows, 1);
35
36          % calcular os caminhos mais curtos e os atrasos de ida e volta
37          for n = 1:nFlows
38              if T(n, 1) == 1 || T(n, 1) == 2
39                  [shortestPath, totalCost] = kShortestPath(D, T(n, 2),
                        T(n, 3), 1);
40                  sP{n} = shortestPath;
41                  delays(n) = totalCost;
42                  Taux(n, :) = T(n, 2:5);
43              elseif T(n, 1) == 3
44                  if ismember(T(n, 2), anycastNodes)
45                      sP{n} = {T(n, 2)};
46                      nSP{n} = 1;
47                      Taux(n, :) = T(n, 2:5);
48                      Taux(n, 3) = T(n, 2);
49                  else
50                      cost = inf;
51                      Taux(n, :) = T(n, 2:5);
52                      for k = anycastNodes
53                          [shortestPath, totalCost] = kShortestPath(D, T
                                (n, 2), k, 1);
54                          if totalCost < cost
55                              sP{n} = shortestPath;
56                              nSP{n} = 1;
57                              cost = totalCost;
58                              delays(n) = totalCost;
59                              Taux(n, 3) = k;
60                          end
61                      end
62                  end
63              end
64          end
65
66          % calcular as cargas dos links
67          Loads = calculateLinkLoads(nNodes, Links, Taux, sP, ones(
                nFlows, 1));
68
```

```
69          % encontrar a pior carga de link
70          worstLinkLoad = max(max(Loads(:, 3:4)));
71
72          % atualizar os melhores resultados se a carga de link for
                menor
73          if worstLinkLoad < minWorstLinkLoad
74              minWorstLinkLoad = worstLinkLoad;
75              bestAnycastNodes = anycastNodes;
76              bestDelays = delays;
77              bestTaux = Taux;
78              bestSP = sP;
79          end
80      end
81  end
82
83  unicastFlows1 = find(T(:, 1) == 1);
84  unicastFlows2 = find(T(:, 1) == 2);
85  anycastFlows = find(T(:, 1) == 3);
86
87  maxDelayUnicast1 = max(bestDelays(unicastFlows1)) * 2 * 1000;
88  avgDelayUnicast1 = mean(bestDelays(unicastFlows1)) * 2 * 1000;
89
90  maxDelayUnicast2 = max(bestDelays(unicastFlows2)) * 2 * 1000;
91  avgDelayUnicast2 = mean(bestDelays(unicastFlows2)) * 2 * 1000;
92
93  maxDelayAnycast = max(bestDelays(anycastFlows)) * 2 * 1000;
94  avgDelayAnycast = mean(bestDelays(anycastFlows)) * 2 * 1000;
95
96  % mostrar os resultados
97  fprintf('Best anycast nodes = %d  %d\n', bestAnycastNodes(1),
        bestAnycastNodes(2));
98  fprintf('Worst link load = %.2f Gbps\n', minWorstLinkLoad);
99  fprintf('Worst round-trip delay (unicast service 1) = %.2f ms\n',
        maxDelayUnicast1);
100 fprintf('Average round-trip delay (unicast service 1) = %.2f ms\n',
        avgDelayUnicast1);
101 fprintf('Worst round-trip delay (unicast service 2) = %.2f ms\n',
        maxDelayUnicast2);
102 fprintf('Average round-trip delay (unicast service 2) = %.2f ms\n',
        avgDelayUnicast2);
103 fprintf('Worst round-trip delay (anycast service) = %.2f ms\n',
        maxDelayAnycast);
104 fprintf('Average round-trip delay (anycast service) = %.2f ms\n',
        avgDelayAnycast);
```

O código desenvolvido no *exercício 1.c.* tem como objetivo encontrar os melhores nós *anycast* para minimizar a carga de link máxima e calcular os atrasos de ida e volta para os diferentes serviços (*unicast* e *anycast*). A estratégia utilizada é testar todas as combinações possíveis de dois nós *anycast* e, para cada par, calcular os caminhos mais curtos, os atrasos de ida e volta e as cargas nos links da rede. O par de nós *anycast* que resulta na menor carga máxima nos links é selecionado.

1.3.2 Resultados e Conclusões

A partir da execução do código, foram obtidos os resultados da figura abaixo (Figura 1.3).

```
------------------------------ Task 1.c.------------------------------
Best anycast nodes = 1   6
Worst link load = 76.60 Gbps
Worst round-trip delay (unicast service 1) = 9.04 ms
Average round-trip delay (unicast service 1) = 5.42 ms
Worst round-trip delay (unicast service 2) = 11.07 ms
Average round-trip delay (unicast service 2) = 5.83 ms
Worst round-trip delay (anycast service) = 6.41 ms
Average round-trip delay (anycast service) = 3.02 ms
```

Figura 1.3: Resultados correspondentes ao exercício 1.c.

A escolha dos nós *anycast* foi crucial para o desempenho da rede. A combinação de nós 1 e 6 foi identificada como a melhor, minimizando a pior carga de link, que foi de 76.60 Gbps. Este valor de carga de link é relativamente baixo, indicando que a distribuição dos fluxos de dados na rede foi otimizada para evitar congestionamentos.

O serviço *anycast* apresentou um desempenho superior ao *unicast* (como esperado), com a pior latência de ida e volta sendo de 6.41 ms e a média de 3.02 ms. A escolha do nó *anycast*, ao minimizar a carga nos links, também ajudou a reduzir os atrasos globais, o que é esperado quando se utiliza qualquer uma das estratégias de encaminhamento *anycast*.

Esses resultados estão de acordo com a expectativa, uma vez que o serviço *anycast* geralmente tende a apresentar menores atrasos devido à sua característica de redirecionar os fluxos para o nó mais próximo em termos de custo de rede. A análise da carga de link também corrobora a eficácia da abordagem adotada para otimizar a rede, distribuindo as cargas de forma equilibrada e reduzindo os congestionamentos.

Em conclusão, o código e a abordagem adotada proporcionaram uma solução eficiente para o problema proposto, otimizando tanto a carga de rede quanto os atrasos para os diferentes tipos de serviço.

1.4 Exercício 1.d.

1.4.1 Código

```
1  %% 1.d.
2
3  clear
4  clc
5
6  fprintf('------------------------------ Task 1.d
       .------------------------------\n');
7
8  % carregar os dados
9  load('InputDataProject2.mat');
```

```
10
11  % par metros
12  nNodes = size(Nodes, 1);
13  nFlows = size(T, 1);
14  nLinks = size(Links, 1);
15
16  v = 2 * 10^5;
17  D = L / v;
18
19  % inicializar vari veis para os melhores resultados
20  bestAnycastNodes = [];
21  minWorstRoundTripDelay = inf;
22  bestDelays = [];
23  bestTaux = [];
24  bestSP = [];
25
26  % testar todas as combina  es poss veis de dois n  s
27  for i = 1:nNodes
28      for j = i+1:nNodes
29          anycastNodes = [i j];
30
31          % inicializar vari veis
32          Taux = zeros(nFlows, 4);
33          delays = zeros(nFlows, 1);
34          sP = cell(nFlows, 1);
35
36          % calcular os caminhos mais curtos e os atrasos de ida e volta
37          for n = 1:nFlows
38              if T(n, 1) == 1 || T(n, 1) == 2
39                  [shortestPath, totalCost] = kShortestPath(D, T(n, 2),
                        T(n, 3), 1);
40                  sP{n} = shortestPath;
41                  delays(n) = totalCost;
42                  Taux(n, :) = T(n, 2:5);
43              elseif T(n, 1) == 3
44                  if ismember(T(n, 2), anycastNodes)
45                      sP{n} = {T(n, 2)};
46                      nSP{n} = 1;
47                      Taux(n, :) = T(n, 2:5);
48                      Taux(n, 3) = T(n, 2);
49                  else
50                      cost = inf;
51                      Taux(n, :) = T(n, 2:5);
52                      for k = anycastNodes
53                          [shortestPath, totalCost] = kShortestPath(D, T
                                (n, 2), k, 1);
54                          if totalCost < cost
55                              sP{n} = shortestPath;
56                              nSP{n} = 1;
57                              cost = totalCost;
58                              delays(n) = totalCost;
59                              Taux(n, 3) = k;
60                          end
61                      end
62                  end
63              end
64          end
65
66          % calcular os atrasos de ida e volta
67          maxDelayAnycast = max(delays(find(T(:, 1) == 3))) * 2 * 1000;
68
```

```
69          % atualizar os melhores resultados se o atraso de ida e volta
                for menor
70          if maxDelayAnycast < minWorstRoundTripDelay
71              minWorstRoundTripDelay = maxDelayAnycast;
72              bestAnycastNodes = anycastNodes;
73              bestDelays = delays;
74              bestTaux = Taux;
75              bestSP = sP;
76          end
77      end
78  end
79
80  % calcular as cargas dos links
81  Loads = calculateLinkLoads(nNodes, Links, bestTaux, bestSP, ones(
        nFlows, 1));
82
83  % encontrar a pior carga de link
84  worstLinkLoad = max(max(Loads(:, 3:4)));
85
86  % Calcular os atrasos de ida e volta
87  maxDelayUnicast1 = max(bestDelays(find(T(:, 1) == 1))) * 2 * 1000;
88  avgDelayUnicast1 = mean(bestDelays(find(T(:, 1) == 1))) * 2 * 1000;
89  maxDelayUnicast2 = max(bestDelays(find(T(:, 1) == 2))) * 2 * 1000;
90  avgDelayUnicast2 = mean(bestDelays(find(T(:, 1) == 2))) * 2 * 1000;
91  maxDelayAnycast = max(bestDelays(find(T(:, 1) == 3))) * 2 * 1000;
92  avgDelayAnycast = mean(bestDelays(find(T(:, 1) == 3))) * 2 * 1000;
93
94  % mostrar os resultados
95  fprintf('Best anycast nodes = %d  %d\n', bestAnycastNodes(1),
        bestAnycastNodes(2));
96  fprintf('Worst link load = %.2f Gbps\n', worstLinkLoad);
97  fprintf('Worst round-trip delay (unicast service 1) = %.2f ms\n',
        maxDelayUnicast1);
98  fprintf('Average round-trip delay (unicast service 1) = %.2f ms\n',
        avgDelayUnicast1);
99  fprintf('Worst round-trip delay (unicast service 2) = %.2f ms\n',
        maxDelayUnicast2);
100 fprintf('Average round-trip delay (unicast service 2) = %.2f ms\n',
        avgDelayUnicast2);
101 fprintf('Worst round-trip delay (anycast service) = %.2f ms\n',
        maxDelayAnycast);
102 fprintf('Average round-trip delay (anycast service) = %.2f ms\n',
        avgDelayAnycast);
```

O código desenvolvido tem como objetivo selecionar os melhores nós *anycast* para o serviço *anycast*, de forma a minimizar o pior atraso de ida e volta da comunicação. O processo foi realizado testando todas as combinações possíveis de dois nós e, para cada combinação, calculando os atrasos de ida e volta dos fluxos.

1.4.2 Resultados e Conclusões

Após executar o código foram obtidos os resultados apresentados na figura abaixo (Figura 1.4).

A combinação de nós 4 e 12 foi escolhida como a melhor, pois resultou no menor pior atraso de ida e volta para o serviço *anycast*, que foi de 4.42 ms. Além disso, a média de atraso para o serviço *anycast* foi de 2.90 ms, o

```
------------------------------ Task 1.d.------------------------------
Best anycast nodes = 4  12
Worst link load = 76.60 Gbps
Worst round-trip delay (unicast service 1) = 9.04 ms
Average round-trip delay (unicast service 1) = 5.42 ms
Worst round-trip delay (unicast service 2) = 11.07 ms
Average round-trip delay (unicast service 2) = 5.83 ms
Worst round-trip delay (anycast service) = 4.42 ms
Average round-trip delay (anycast service) = 2.90 ms
```

Figura 1.4: Resultados correspondentes ao exercício 1.d.

que indica um bom desempenho para essa configuração.

A carga de link máxima foi de 76.60 Gbps tal como no *exercício 1.c.* (Seção 1.3), o que sugere que a distribuição de tráfego entre os nós foi eficaz, mas a carga de link não foi otimizada em termos de balanceamento total, visto que o objetivo principal aqui foi minimizar o atraso e não a carga.

Em conclusão, os resultados obtidos estão de acordo com as expectativas, demonstrando que a escolha dos nós *anycast* minimiza significativamente os atrasos de ida e volta, proporcionando um serviço de maior qualidade em comparação com os serviços *unicast*. A análise de carga de link confirma que o tráfego foi distribuído de maneira eficiente, embora uma maior ênfase no balanceamento de carga poderia ter resultado em uma distribuição mais equilibrada dos fluxos de rede.

1.5 Exercício 1.e.

1.5.1 Código

```
1  %% 1.e.
2
3  clear
4  clc
5
6  fprintf('------------------------------ Task 1.e
       .------------------------------\n');
7
8  % carregar os dados
9  load('InputDataProject2.mat');
10
11 % par metros
12 nNodes = size(Nodes, 1);
13 nFlows = size(T, 1);
14 nLinks = size(Links, 1);
15
16 v = 2 * 10^5;
17 D = L / v;
18
19 % inicializar vari veis para os melhores resultados
```

```
bestAnycastNodes = [];
minAvgRoundTripDelay = inf;
bestDelays = [];
bestTaux = [];
bestSP = [];

% testar todas as combina  es poss veis de dois n s
for i = 1:nNodes
    for j = i+1:nNodes
        anycastNodes = [i j];

        % inicializar vari veis
        Taux = zeros(nFlows, 4);
        delays = zeros(nFlows, 1);
        sP = cell(nFlows, 1);

        % calcular os caminhos mais curtos e os atrasos de ida e volta
        for n = 1:nFlows
            if T(n, 1) == 1 || T(n, 1) == 2
                [shortestPath, totalCost] = kShortestPath(D, T(n, 2),
                    T(n, 3), 1);
                sP{n} = shortestPath;
                delays(n) = totalCost;
                Taux(n, :) = T(n, 2:5);
            elseif T(n, 1) == 3
                if ismember(T(n, 2), anycastNodes)
                    sP{n} = {T(n, 2)};
                    nSP{n} = 1;
                    Taux(n, :) = T(n, 2:5);
                    Taux(n, 3) = T(n, 2);
                else
                    cost = inf;
                    Taux(n, :) = T(n, 2:5);
                    for k = anycastNodes
                        [shortestPath, totalCost] = kShortestPath(D, T
                            (n, 2), k, 1);
                        if totalCost < cost
                            sP{n} = shortestPath;
                            nSP{n} = 1;
                            cost = totalCost;
                            delays(n) = totalCost;
                            Taux(n, 3) = k;
                        end
                    end
                end
            end
        end

        % calcular os atrasos de ida e volta
        avgDelayAnycast = mean(delays(find(T(:, 1) == 3))) * 2 * 1000;

        % atualizar os melhores resultados se o atraso m dio de ida e
            volta for menor
        if avgDelayAnycast < minAvgRoundTripDelay
            minAvgRoundTripDelay = avgDelayAnycast;
            bestAnycastNodes = anycastNodes;
            bestDelays = delays;
            bestTaux = Taux;
            bestSP = sP;
        end
    end
end
```

```
79
80  % calcular as cargas dos links
81  Loads = calculateLinkLoads(nNodes, Links, bestTaux, bestSP, ones(
        nFlows, 1));
82
83  % encontrar a pior carga de link
84  worstLinkLoad = max(max(Loads(:, 3:4)));
85
86  % calcular os atrasos de ida e volta
87  maxDelayUnicast1 = max(bestDelays(find(T(:, 1) == 1))) * 2 * 1000;
88  avgDelayUnicast1 = mean(bestDelays(find(T(:, 1) == 1))) * 2 * 1000;
89  maxDelayUnicast2 = max(bestDelays(find(T(:, 1) == 2))) * 2 * 1000;
90  avgDelayUnicast2 = mean(bestDelays(find(T(:, 1) == 2))) * 2 * 1000;
91  maxDelayAnycast = max(bestDelays(find(T(:, 1) == 3))) * 2 * 1000;
92  avgDelayAnycast = mean(bestDelays(find(T(:, 1) == 3))) * 2 * 1000;
93
94  % mostrar os resultados
95  fprintf('Best anycast nodes = %d  %d\n', bestAnycastNodes(1),
        bestAnycastNodes(2));
96  fprintf('Worst link load = %.2f Gbps\n', worstLinkLoad);
97  fprintf('Worst round-trip delay (unicast service 1) = %.2f ms\n',
        maxDelayUnicast1);
98  fprintf('Average round-trip delay (unicast service 1) = %.2f ms\n',
        avgDelayUnicast1);
99  fprintf('Worst round-trip delay (unicast service 2) = %.2f ms\n',
        maxDelayUnicast2);
100 fprintf('Average round-trip delay (unicast service 2) = %.2f ms\n',
        avgDelayUnicast2);
101 fprintf('Worst round-trip delay (anycast service) = %.2f ms\n',
        maxDelayAnycast);
102 fprintf('Average round-trip delay (anycast service) = %.2f ms\n',
        avgDelayAnycast);
```

O objetivo deste exercício foi encontrar a combinação de dois nós para o serviço *anycast* que minimiza o atraso médio de ida e volta.

1.5.2 Resultados e Conclusões

```
------------------------------ Task 1.e.------------------------------
Best anycast nodes = 5  14
Worst link load = 76.60 Gbps
Worst round-trip delay (unicast service 1) = 9.04 ms
Average round-trip delay (unicast service 1) = 5.42 ms
Worst round-trip delay (unicast service 2) = 11.07 ms
Average round-trip delay (unicast service 2) = 5.83 ms
Worst round-trip delay (anycast service) = 4.90 ms
Average round-trip delay (anycast service) = 2.52 ms
```

Figura 1.5: Resultados correspondentes ao exercício 1.e.

A combinação ideal foi encontrada nos nós 5 e 14, resultando em um atraso médio de ida e volta de 2.52 ms tal como se pode ver na Figura 1.5.

Em comparação com os serviços *unicast*, o serviço *anycast* apresenta um

desempenho superior. O pior atraso de ida e volta para o serviço *anycast* foi de 4.90 ms. A média de atrasos também favorece o serviço *anycast*, com uma média de 2.52 ms, em comparação com as médias de 5.42 ms e 5.83 ms para os serviços *unicast*.

Em resumo, a escolha dos nós *anycast* que minimizam o atraso médio de ida e volta proporciona um serviço com latência muito inferior em comparação aos serviços *unicast*. Isso comprova a eficácia do modelo *anycast* em redes de dados, principalmente quando a latência é um fator crítico. A análise de carga de link sugere que o balanceamento de tráfego não foi uma preocupação principal neste exercício, mas ainda assim o desempenho foi satisfatório.

1.6 Exercício 1.f.

Ao comparar os resultados obtidos nos diferentes exercícios, é possível observar algumas tendências e conclusões importantes:

- O serviço *anycast* consistentemente apresenta um desempenho superior em termos de latência em comparação com os serviços *unicast*. Em todos os exercícios, o pior atraso de ida e volta para o serviço *anycast* é menor do que o pior atraso de ida e volta nos serviços *unicast*, e a média de atrasos também favorece o *anycast*.

- A combinação ótima de nós *anycast* varia entre os exercícios. Por exemplo, nos exercícios 1.a., 1.d. e 1.c., os melhores nós foram 3 e 10, 4 e 12, e 1 e 6, respectivamente. Isso sugere que a escolha dos nós depende de diferentes condições e critérios (como o pior atraso ou a média de atraso), mas o padrão geral é que o *anycast* oferece menores atrasos em comparação aos serviços *unicast*.

- Em termos de latência, o exercício 1.e. obteve o menor atraso médio de ida e volta (*anycast*: 2.52 ms), o que indica que a combinação dos nós 5 e 14 é a mais eficiente para minimizar a latência geral da rede.

- De forma geral, os resultados sugerem que o modelo *anycast* é eficiente para reduzir a latência de comunicação.

Funções Auxiliares

Neste capítulo, são descritas as funções auxiliares utilizadas para a implementação e otimização das soluções dos problema seguintes proposto. As funções abordadas são a `greedyRandomizedStrategy` e a `HillClimbingStrategy`, que desempenham papéis fundamentais na geração de soluções iniciais e na subsequente melhoria destas, respetivamente.

Função `greedyRandomizedStrategy`

```
function [sol, load] = greedyRandomizedStrategy(nNodes, Links, T, sP,
    nSP)
    nFlows = size(T,1);
    % ordem aleat ria dos fluxos
    randFlows = randperm(nFlows);
    sol = zeros(1, nFlows);

    % iterar por cada fluxo
    for flow = randFlows
        path_index = 0;
        best_load = inf;

        % testar cada caminho "poss vel" com uma certa carga
        for path = 1 : nSP(flow)
            % tentar o caminho para esse fluxo
            sol(flow) = path;
            % calcular as cargas
            Loads = calculateLinkLoads(nNodes, Links, T, sP, sol);
            load = max(max(Loads(:, 3:4)));

            % verificar se a carga atual    melhor que a melhor carga
            if load < best_load
                % change index of path and load
                path_index = path;
                best_load = load;
            end
        end
        sol(flow) = path_index;
    end
    load = best_load;
end
```

A função `greedyRandomizedStrategy` é responsável pela geração de soluções iniciais de forma "greedy randomized" em português, de forma "gananciosa aleatória". Este método combina aleatoriedade com uma abordagem

determinística na seleção dos caminhos para os fluxos de tráfego, o que assegura soluções viáveis com cargas reduzidas. O algoritmo segue os seguintes passos:

- Determina uma ordem aleatória para os fluxos de tráfego, e garante diversificação nas soluções iniciais.
- Para cada fluxo, testa todas as rotas candidatas permitidas pelo algoritmo de caminhos mais curtos.
- Para cada rota, calcula as cargas nas ligações através da função `calculateLinkLoads` e determina a rota que minimiza a carga máxima.
- Armazena a rota escolhida para cada fluxo, bem como a carga máxima observada na rede.

O principal objetivo desta função é fornecer uma solução inicial para algoritmos subsequentes de otimização local, como o `HillClimbingStrategy`. A abordagem aleatória evita soluções únicas, promovendo assim a exploração de diferentes soluções.

Função `HillClimbingStrategy`

```
function [sol, load] = HillClimbingStrategy(nNodes, Links, T, sP, nSP,
     sol, load)
    nFlows = size(T,1);
    % definir as melhores vari veis locais
    bestLocalLoad = load;
    bestLocalSol = sol;

    % hill climbing
    improved = true;
    while improved

        % testar cada fluxo
        for flow = 1 : nFlows
            % testar cada caminho do fluxo
            for path = 1 : nSP(flow)
                if path ~= sol(flow)

                    % alterar o caminho para esse fluxo
                    auxSol = sol;
                    auxSol(flow) = path;

                    % calcular as cargas
                    Loads = calculateLinkLoads(nNodes, Links, T, sP,
                        auxSol);
                    auxLoad = max(max(Loads(:, 3:4)));

                    % verificar se a carga atual    melhor que a carga
                         inicial
                    if auxLoad < bestLocalLoad
                        bestLocalLoad = auxLoad;
                        bestLocalSol = auxSol;
```

```
29                      end
30                  end
31              end
32          end
33
34          if bestLocalLoad < load
35              load = bestLocalLoad;
36              sol = bestLocalSol;
37          else
38              improved = false;
39          end
40      end
41  end
```

A função `HillClimbingStrategy` implementa uma abordagem iterativa de otimização local, baseada no método de hill climbing (*"subida de colina"*). Esta função parte de uma solução inicial fornecida e tenta melhorá-la iterativamente até atingir um ponto de ótimo local. Os principais passos do algoritmo são os seguintes:

- Para cada fluxo de tráfego, testa todas as rotas candidatas disponíveis, exceto a rota atual.
- Calcula a carga nas ligações resultante da alteração de rota para o fluxo em questão, utilizando a função `calculateLinkLoads`.
- Se a nova solução apresentar uma carga máxima inferior à da solução atual, atualiza a solução e continua o processo.
- O algoritmo termina quando não é possível melhorar a solução atual.

A função `HillClimbingStrategy` é fundamental para refinar as soluções iniciais e minimizar a carga máxima nas ligações da rede. Apesar de poder convergir para ótimos locais, a combinação com métodos como `greedyRandomizedStrategy` garante diversificação e melhora a qualidade global das soluções encontradas.

Capítulo 2

Task 2

2.1 Exercício 2.a.

2.1.1 Código

```
%% 2.a.

clear
clc

fprintf('------------------------------ Task 2.a
    .------------------------------\n');

% carregar os dados
load('InputDataProject2.mat');

% par metros
nNodes = size(Nodes, 1);
nFlows = size(T, 1);
nLinks = size(Links, 1);
k = 6;

v = 2 * 10^5;
D = L / v;

anycastNodes = [3 10];

% inicializar vari veis para os atrasos e caminhos
Taux = zeros(nFlows, 4);
delays = zeros(nFlows, 1);
sP = cell(nFlows, 1);
nSP = zeros(nFlows, 1);

% calcular os caminhos mais curtos e os atrasos de ida e volta
for n = 1:nFlows
    if T(n, 1) == 1 || T(n, 1) == 2
        [shortestPaths, totalCosts] = kShortestPath(D, T(n, 2), T(n,
            3), k);
        sP{n} = shortestPaths;
        nSP(n) = length(shortestPaths);
        delays(n) = totalCosts(1);
        Taux(n, :) = T(n, 2:5);
    elseif T(n, 1) == 3
```

```
37          if ismember(T(n, 2), anycastNodes)
38              sP{n} = {T(n, 2)};
39              nSP(n) = 1;
40              Taux(n, :) = T(n, 2:5);
41              Taux(n, 3) = T(n, 2);
42          else
43              cost = inf;
44              Taux(n, :) = T(n, 2:5);
45              for i = anycastNodes
46                  [shortestPaths, totalCosts] = kShortestPath(D, T(n, 2)
                        , i, k);
47                  if totalCosts(1) < cost
48                      sP{n} = shortestPaths;
49                      nSP(n) = length(shortestPaths);
50                      cost = totalCosts(1);
51                      delays(n) = totalCosts(1);
52                      Taux(n, 3) = i;
53                  end
54              end
55          end
56      end
57  end
58
59  unicastFlows1 = find(T(:, 1) == 1);
60  unicastFlows2 = find(T(:, 1) == 2);
61  anycastFlows = find(T(:, 1) == 3);
62
63  maxDelayUnicast1 = max(delays(unicastFlows1)) * 2 * 1000;
64  avgDelayUnicast1 = mean(delays(unicastFlows1)) * 2 * 1000;
65
66  maxDelayUnicast2 = max(delays(unicastFlows2)) * 2 * 1000;
67  avgDelayUnicast2 = mean(delays(unicastFlows2)) * 2 * 1000;
68
69  maxDelayAnycast = max(delays(anycastFlows)) * 2 * 1000;
70  avgDelayAnycast = mean(delays(anycastFlows)) * 2 * 1000;
71
72  % par metros do algoritmo
73  maxTime = 30;
74  bestLoad = inf;
75  bestSol = [];
76  totalCycles = 0;
77  bestCycle = 0;
78  startTime = tic;
79
80  % "correr" o algoritmo Multi Start Hill Climbing
81  while toc(startTime) < maxTime
82      % solu  o inicial
83      [sol, load] = greedyRandomizedStrategy(nNodes, Links, Taux, sP,
            nSP);
84
85      % melhorar a solu  o inicial
86      [sol, load] = HillClimbingStrategy(nNodes, Links, Taux, sP, nSP,
            sol, load);
87
88      totalCycles = totalCycles + 1;
89
90      % verificar se a solu  o atual    a melhor encontrada
91      if load < bestLoad
92          bestLoad = load;
93          bestSol = sol;
94          bestTime = toc(startTime);
95          bestCycle = totalCycles;
```

```
96      end
97  end
98
99  % mostrar os resultados
100 fprintf('Multi Start Hill Climbing algorithm with initial Greedy
        Randomized (Anycast nodes: %d %d)\n', anycastNodes(1),
        anycastNodes(2));
101 fprintf('Worst link load of the network: %.2f\n', bestLoad);
102 fprintf('Total number of cycles run: %d\n', totalCycles);
103 fprintf('Running time at which the best solution was obtained: %.2f
        seconds\n', bestTime);
104 fprintf('Number of cycles at which the best solution was obtained: %d\
        n', bestCycle);
```

O objetivo do exercício 2.a. é minimizar a pior carga do link na rede através do algoritmo Multi Start Hill Climbing com soluções inicializadas por Greedy Randomized. Esta task considera os nós anycast 3 e 10, com tempo de execução de 30 segundos e $k = 6$ caminhos candidatos para cada fluxo unicast.

2.1.2 Resultados e Conclusões

```
------------------------------ Task 2.a.------------------------------
Multi Start Hill Climbing algorithm with initial Greedy Randomized (Anycast nodes: 3 10)
Worst link load of the network: 64.20
Total number of cycles run: 2669
Running time at which the best solution was obtained: 8.61 seconds
Number of cycles at which the best solution was obtained: 773
```

Figura 2.1: Resultados correspondentes ao exercício 2.a.

Com base nos resultados obtidos concluímos que o algoritmo foi capaz de encontrar uma solução eficiente dentro do tempo limite fornecido. O valor da *Worst link load* é de 64.20 Gbps e é satisfatório no contexto de otimização da rede. A convergência para a melhor solução ocorreu relativamente cedo (após 8.61 segundos e 773 ciclos), o que demonstra a eficácia do algoritmo. Por fim, conclui-se também que os nós anycast definidos (3 e 10) conseguem proporcionar um equilíbrio adequado entre a distribuição do tráfego e a minimização da carga nos enlaces.

2.2 Exercício 2.b.

2.2.1 Código

```
1  %% 2.b.
2
3  clear
4
5  fprintf('------------------------------ Task 2.b
       .------------------------------\n');
```

```
6
7  % carregar os dados
8  load('InputDataProject2.mat');
9
10 % par metros
11 nNodes = size(Nodes, 1);
12 nFlows = size(T, 1);
13 nLinks = size(Links, 1);
14 k = 6;
15
16 v = 2 * 10^5;
17 D = L / v;
18
19 anycastNodes = [1 6];
20
21 % inicializar vari veis para os atrasos e caminhos
22 Taux = zeros(nFlows, 4);
23 delays = zeros(nFlows, 1);
24 sP = cell(nFlows, 1);
25 nSP = zeros(nFlows, 1);
26
27 % calcular os caminhos mais curtos e os atrasos de ida e volta
28 for n = 1:nFlows
29     if T(n, 1) == 1 || T(n, 1) == 2
30         [shortestPaths, totalCosts] = kShortestPath(D, T(n, 2), T(n,
               3), k);
31         sP{n} = shortestPaths;
32         nSP(n) = length(shortestPaths);
33         delays(n) = totalCosts(1);
34         Taux(n, :) = T(n, 2:5);
35     elseif T(n, 1) == 3
36         if ismember(T(n, 2), anycastNodes)
37             sP{n} = {T(n, 2)};
38             nSP(n) = 1;
39             Taux(n, :) = T(n, 2:5);
40             Taux(n, 3) = T(n, 2);
41         else
42             cost = inf;
43             Taux(n, :) = T(n, 2:5);
44             for i = anycastNodes
45                 [shortestPaths, totalCosts] = kShortestPath(D, T(n, 2)
                       , i, k);
46                 if totalCosts(1) < cost
47                     sP{n} = shortestPaths;
48                     nSP(n) = length(shortestPaths);
49                     cost = totalCosts(1);
50                     delays(n) = totalCosts(1);
51                     Taux(n, 3) = i;
52                 end
53             end
54         end
55     end
56 end
57
58 unicastFlows1 = find(T(:, 1) == 1);
59 unicastFlows2 = find(T(:, 1) == 2);
60 anycastFlows = find(T(:, 1) == 3);
61
62 maxDelayUnicast1 = max(delays(unicastFlows1)) * 2 * 1000;
63 avgDelayUnicast1 = mean(delays(unicastFlows1)) * 2 * 1000;
64
65 maxDelayUnicast2 = max(delays(unicastFlows2)) * 2 * 1000;
```

```
avgDelayUnicast2 = mean(delays(unicastFlows2)) * 2 * 1000;

maxDelayAnycast = max(delays(anycastFlows)) * 2 * 1000;
avgDelayAnycast = mean(delays(anycastFlows)) * 2 * 1000;

% par metros do algoritmo
maxTime = 30;
bestLoad = inf;
bestSol = [];
totalCycles = 0;
bestCycle = 0;
startTime = tic;

% "correr" o algoritmo Multi Start Hill Climbing
while toc(startTime) < maxTime
    % solu   o inicial
    [sol, load] = greedyRandomizedStrategy(nNodes, Links, Taux, sP,
        nSP);

    % melhorar a solu   o inicial
    [sol, load] = HillClimbingStrategy(nNodes, Links, Taux, sP, nSP,
        sol, load);

    totalCycles = totalCycles + 1;

    % verificar se a solu   o atual    a melhor encontrada
    if load < bestLoad
        bestLoad = load;
        bestSol = sol;
        bestTime = toc(startTime);
        bestCycle = totalCycles;
    end
end

% mostrar os resultados
fprintf('Multi Start Hill Climbing algorithm with initial Greedy
    Randomized (Anycast nodes: %d %d)\n', anycastNodes(1),
    anycastNodes(2));
fprintf('Worst link load of the network: %.2f\n', bestLoad);
fprintf('Total number of cycles run: %d\n', totalCycles);
fprintf('Running time at which the best solution was obtained: %.2f
    seconds\n', bestTime);
fprintf('Number of cycles at which the best solution was obtained: %d\
    n', bestCycle);
```

O objetivo da task 2.b é utilizar o algoritmo *Multi Start Hill Climbing* com uma solução inicial baseada na estratégia *Greedy Randomized*, com os nós anycast 1 e 6. Esta task visa minimizar a pior carga de link na rede durante a execução do algoritmo.

2.2.2 Resultados e Conclusões

Os resultados evidenciam que o algoritmo foi eficiente em encontrar a melhor solução em um tempo extremamente curto (0.04 segundos) e em apenas 2 ciclos de execução. Isso demonstra que a configuração inicial escolhida foi robusta e que o algoritmo conseguiu rapidamente melhorar a distribuição de tráfego na rede, minimizando a carga máxima dos links.

```
------------------------------ Task 2.b.------------------------------
Multi Start Hill Climbing algorithm with initial Greedy Randomized (Anycast nodes: 1 6)
Worst link load of the network: 60.20
Total number of cycles run: 3274
Running time at which the best solution was obtained: 0.04 seconds
Number of cycles at which the best solution was obtained: 2
```

Figura 2.2: Resultados correspondentes ao exercício 2.b.

A pior carga de link de 60.20 Gbps representa um cenário otimizado, o que mostra que a abordagem implementada é eficaz para redistribuir os fluxos de tráfego de modo a conseguir reduzir o impacto em enlaces sobrecarregados.

2.3 Exercício 2.c.

2.3.1 Código

```
%% 2.c.

clear

fprintf('------------------------------ Task 2.c
    .------------------------------\n');

% carregar os dados
load('InputDataProject2.mat');

% par metros
nNodes = size(Nodes, 1);
nFlows = size(T, 1);
nLinks = size(Links, 1);
k = 6;

v = 2 * 10^5;
D = L / v;

anycastNodes = [4 12];

% inicializar vari veis para os atrasos e caminhos
Taux = zeros(nFlows, 4);
delays = zeros(nFlows, 1);
sP = cell(nFlows, 1);
nSP = zeros(nFlows, 1);

% calcular os caminhos mais curtos e os atrasos de ida e volta
for n = 1:nFlows
    if T(n, 1) == 1 || T(n, 1) == 2
        [shortestPaths, totalCosts] = kShortestPath(D, T(n, 2), T(n,
            3), k);
        sP{n} = shortestPaths;
        nSP(n) = length(shortestPaths);
        delays(n) = totalCosts(1);
        Taux(n, :) = T(n, 2:5);
    elseif T(n, 1) == 3
        if ismember(T(n, 2), anycastNodes)
```

```
37              sP{n} = {T(n, 2)};
38              nSP(n) = 1;
39              Taux(n, :) = T(n, 2:5);
40              Taux(n, 3) = T(n, 2);
41          else
42              cost = inf;
43              Taux(n, :) = T(n, 2:5);
44              for i = anycastNodes
45                  [shortestPaths, totalCosts] = kShortestPath(D, T(n, 2)
                        , i, k);
46                  if totalCosts(1) < cost
47                      sP{n} = shortestPaths;
48                      nSP(n) = length(shortestPaths);
49                      cost = totalCosts(1);
50                      delays(n) = totalCosts(1);
51                      Taux(n, 3) = i;
52                  end
53              end
54          end
55      end
56  end
57
58  unicastFlows1 = find(T(:, 1) == 1);
59  unicastFlows2 = find(T(:, 1) == 2);
60  anycastFlows = find(T(:, 1) == 3);
61
62  maxDelayUnicast1 = max(delays(unicastFlows1)) * 2 * 1000;
63  avgDelayUnicast1 = mean(delays(unicastFlows1)) * 2 * 1000;
64
65  maxDelayUnicast2 = max(delays(unicastFlows2)) * 2 * 1000;
66  avgDelayUnicast2 = mean(delays(unicastFlows2)) * 2 * 1000;
67
68  maxDelayAnycast = max(delays(anycastFlows)) * 2 * 1000;
69  avgDelayAnycast = mean(delays(anycastFlows)) * 2 * 1000;
70
71  % par metros do algoritmo
72  maxTime = 30;
73  bestLoad = inf;
74  bestSol = [];
75  totalCycles = 0;
76  bestCycle = 0;
77  startTime = tic;
78
79  % "correr" o algoritmo Multi Start Hill Climbing
80  while toc(startTime) < maxTime
81      % solu  o inicial
82      [sol, load] = greedyRandomizedStrategy(nNodes, Links, Taux, sP,
            nSP);
83
84      % melhorar a solu  o inicial
85      [sol, load] = HillClimbingStrategy(nNodes, Links, Taux, sP, nSP,
            sol, load);
86
87      totalCycles = totalCycles + 1;
88
89      % verificar se a solu  o atual    a melhor encontrada
90      if load < bestLoad
91          bestLoad = load;
92          bestSol = sol;
93          bestTime = toc(startTime);
94          bestCycle = totalCycles;
95      end
```

```
96  end
97
98  % mostrar os resultados
99  fprintf('Multi Start Hill Climbing algorithm with initial Greedy
        Randomized (Anycast nodes: %d %d)\n', anycastNodes(1),
        anycastNodes(2));
100 fprintf('Worst link load of the network: %.2f\n', bestLoad);
101 fprintf('Total number of cycles run: %d\n', totalCycles);
102 fprintf('Running time at which the best solution was obtained: %.2f
        seconds\n', bestTime);
103 fprintf('Number of cycles at which the best solution was obtained: %d\
        n', bestCycle);
```

Tal como na task anterior, o objetivo desta é utilizar o *Multi Start Hill Climbing* utilizando *Greedy Randomized*. Os nós anycast selecionados foram os mesmos das task 1.d. (nós 4 e 12), com o intuito de minimizar a pior carga de link na rede.

2.3.2 Resultados e Conclusões

Nestes resultados algoritmo *Multi Start Hill Climbing* demonstrou ser eficaz em encontrar a melhor solução com uma carga de link mínima de 60.00 Gbps, o que representa uma otimização na utilização da capacidade da rede relativamente aos nós já analisados.

```
------------------------------ Task 2.c.------------------------------
Multi Start Hill Climbing algorithm with initial Greedy Randomized (Anycast nodes: 4 12)
Worst link load of the network: 60.00
Total number of cycles run: 2494
Running time at which the best solution was obtained: 0.11 seconds
Number of cycles at which the best solution was obtained: 7
```

Figura 2.3: Resultados correspondentes ao exercício 2.c.

2.4 Exercício 2.d.

2.4.1 Código

```
1  %% 2.d.
2
3  clear
4
5  fprintf('------------------------------ Task 2.d
       .------------------------------\n');
6
7  % carregar os dados
8  load('InputDataProject2.mat');
9
10 % par metros
11 nNodes = size(Nodes, 1);
12 nFlows = size(T, 1);
13 nLinks = size(Links, 1);
```

```
14 k = 6;
15
16 v = 2 * 10^5;
17 D = L / v;
18
19 anycastNodes = [5 14];
20
21 % inicializar vari veis para os atrasos e caminhos
22 Taux = zeros(nFlows, 4);
23 delays = zeros(nFlows, 1);
24 sP = cell(nFlows, 1);
25 nSP = zeros(nFlows, 1);
26
27 % calcular os caminhos mais curtos e os atrasos de ida e volta
28 for n = 1:nFlows
29     if T(n, 1) == 1 || T(n, 1) == 2
30         [shortestPaths, totalCosts] = kShortestPath(D, T(n, 2), T(n,
              3), k);
31         sP{n} = shortestPaths;
32         nSP(n) = length(shortestPaths);
33         delays(n) = totalCosts(1);
34         Taux(n, :) = T(n, 2:5);
35     elseif T(n, 1) == 3
36         if ismember(T(n, 2), anycastNodes)
37             sP{n} = {T(n, 2)};
38             nSP(n) = 1;
39             Taux(n, :) = T(n, 2:5);
40             Taux(n, 3) = T(n, 2);
41         else
42             cost = inf;
43             Taux(n, :) = T(n, 2:5);
44             for i = anycastNodes
45                 [shortestPaths, totalCosts] = kShortestPath(D, T(n, 2)
                      , i, k);
46                 if totalCosts(1) < cost
47                     sP{n} = shortestPaths;
48                     nSP(n) = length(shortestPaths);
49                     cost = totalCosts(1);
50                     delays(n) = totalCosts(1);
51                     Taux(n, 3) = i;
52                 end
53             end
54         end
55     end
56 end
57
58 unicastFlows1 = find(T(:, 1) == 1);
59 unicastFlows2 = find(T(:, 1) == 2);
60 anycastFlows = find(T(:, 1) == 3);
61
62 maxDelayUnicast1 = max(delays(unicastFlows1)) * 2 * 1000;
63 avgDelayUnicast1 = mean(delays(unicastFlows1)) * 2 * 1000;
64
65 maxDelayUnicast2 = max(delays(unicastFlows2)) * 2 * 1000;
66 avgDelayUnicast2 = mean(delays(unicastFlows2)) * 2 * 1000;
67
68 maxDelayAnycast = max(delays(anycastFlows)) * 2 * 1000;
69 avgDelayAnycast = mean(delays(anycastFlows)) * 2 * 1000;
70
71 % par metros do algoritmo
72 maxTime = 30;
73 bestLoad = inf;
```

```
bestSol = [];
totalCycles = 0;
bestCycle = 0;
startTime = tic;

% "correr" o algoritmo Multi Start Hill Climbing
while toc(startTime) < maxTime
    % solu  o inicial
    [sol, load] = greedyRandomizedStrategy(nNodes, Links, Taux, sP,
        nSP);

    % melhorar a solu  o inicial
    [sol, load] = HillClimbingStrategy(nNodes, Links, Taux, sP, nSP,
        sol, load);

    totalCycles = totalCycles + 1;

    % verificar se a solu  o atual    a melhor encontrada
    if load < bestLoad
        bestLoad = load;
        bestSol = sol;
        bestTime = toc(startTime);
        bestCycle = totalCycles;
    end
end

% mostrar os resultados
fprintf('Multi Start Hill Climbing algorithm with initial Greedy
    Randomized (Anycast nodes: %d %d)\n', anycastNodes(1),
    anycastNodes(2));
fprintf('Worst link load of the network: %.2f\n', bestLoad);
fprintf('Total number of cycles run: %d\n', totalCycles);
fprintf('Running time at which the best solution was obtained: %.2f
    seconds\n', bestTime);
fprintf('Number of cycles at which the best solution was obtained: %d\
    n', bestCycle);
```

2.4.2 Resultados e Conclusões

```
------------------------------ Task 2.d.------------------------------
Multi Start Hill Climbing algorithm with initial Greedy Randomized (Anycast nodes: 5 14)
Worst link load of the network: 62.90
Total number of cycles run: 3542
Running time at which the best solution was obtained: 0.01 seconds
Number of cycles at which the best solution was obtained: 1
```

Figura 2.4: Resultados correspondentes ao exercício 2.d.

Os resultados demonstram que o algoritmo foi capaz de encontrar num curto espaço de tempo uma solução eficiente com uma carga máxima de 62.90 Gbps. A melhor solução foi obtida logo no primeiro ciclo, em apenas 0.01 segundos, indicando que a configuração inicial foi eficaz na aproximação de uma solução otimizada.

2.5 Exercício 2.e.

Esta task tem como objetivo comparar os resultados obtidos em todos os experimentos realizados na Task 2 e na Task1.

Worst Link Load:

- Na Task 1, os resultados foram baseados em rotas fixas determinadas por combinações de nós anycast. A menor carga foi obtida com os nós 1 e 6 (76.60 Gbps).
- Na Task 2, o uso do algoritmo *Multi Start Hill Climbing* reduziu significativamente a carga máxima, alcançando valores mais baixos, 60.00 Gbps (nos nós 4 e 12 da Task 2.c).

Round-trip:

- Na Task 1, o foco principal estava nos atrasos (*worst round-trip delay* e *average round-trip delay*) e apesar de eficiente em minimizar os atrasos (com valores entre 4.42 ms e 9.04 ms), os atrasos médios nos serviços anycast demonstraram-se vantajosos em todas as configurações.
- A Task 2 não apresentou comparações diretas sobre atrasos porque o objetivo principal era a carga de link, contudo como houve melhorias na distribuição de tráfego.

Na Task 2, o uso do algoritmo *Greedy + Hill Climbing* ajudou a diminuir a pior carga de link da rede, uma vez que consegue ajustar melhor o tráfego ao explorar várias opções rapidamente, encontrando boas soluções de forma eficiente. Já na Task 1, o foco foi só nos atrasos, e mesmo funcionando bem, não foi tão bom para redistribuir o tráfego e diminuir o peso em links mais usados. Isso mostra que começar com uma boa estratégia inicial (*Greedy*) e melhorar com ajustes automáticos (*Hill Climbing*) dá resultados bem melhores e mais equilibrados.

Capítulo 3

Task 3

3.1 Exercício 3.a.

3.1.1 Código

```
%% 3.a.

clear
clc

fprintf('------------------------------ Task 3.a
    .------------------------------\n');

% carregar os dados
load('InputDataProject2.mat');

% par metros
nNodes = size(Nodes, 1);
nFlows = size(T, 1);
nLinks = size(Links, 1);
k = 12;

v = 2 * 10^5;
D = L / v;

anycastNodes = [3 10];

% inicializar vari veis para os atrasos e caminhos
Taux = zeros(nFlows, 4);
delays = zeros(nFlows, 1);
sP = cell(nFlows, 1);
nSP = zeros(nFlows, 1);
firstPaths = cell(nFlows, 1);
secondPaths = cell(nFlows, 1);

% calcular os caminhos mais curtos e os atrasos de ida e volta
for n = 1:nFlows
    if T(n, 1) == 1
        [shortestPaths, totalCosts] = kShortestPath(D, T(n, 2), T(n,
            3), k);
        sP{n} = shortestPaths;
        nSP(n) = length(shortestPaths);
        delays(n) = totalCosts(1);
```

```
37          Taux(n, :) = T(n, 2:5);
38      elseif T(n, 1) == 2
39          [firstPaths{n}, secondPaths{n}, totalPairCosts] =
                kShortestPathPairs(D, T(n, 2), T(n, 3), k);
40          sP{n} = firstPaths{n};
41          nSP(n) = length(firstPaths{n});
42          delays(n) = totalPairCosts(1);
43          Taux(n, :) = T(n, 2:5);
44      elseif T(n, 1) == 3
45          if ismember(T(n, 2), anycastNodes)
46              sP{n} = {T(n, 2)};
47              nSP(n) = 1;
48              Taux(n, :) = T(n, 2:5);
49              Taux(n, 3) = T(n, 2);
50          else
51              cost = inf;
52              Taux(n, :) = T(n, 2:5);
53              for i = anycastNodes
54                  [shortestPaths, totalCosts] = kShortestPath(D, T(n, 2)
                        , i, k);
55                  if totalCosts(1) < cost
56                      sP{n} = shortestPaths;
57                      nSP(n) = length(shortestPaths);
58                      cost = totalCosts(1);
59                      delays(n) = totalCosts(1);
60                      Taux(n, 3) = i;
61                  end
62              end
63          end
64      end
65  end
66
67  unicastFlows1 = find(T(:, 1) == 1);
68  unicastFlows2 = find(T(:, 1) == 2);
69  anycastFlows = find(T(:, 1) == 3);
70
71  maxDelayUnicast1 = max(delays(unicastFlows1)) * 2 * 1000;
72  avgDelayUnicast1 = mean(delays(unicastFlows1)) * 2 * 1000;
73
74  maxDelayUnicast2 = max(delays(unicastFlows2)) * 2 * 1000;
75  avgDelayUnicast2 = mean(delays(unicastFlows2)) * 2 * 1000;
76
77  maxDelayAnycast = max(delays(anycastFlows)) * 2 * 1000;
78  avgDelayAnycast = mean(delays(anycastFlows)) * 2 * 1000;
79
80  % par metros do algoritmo
81  maxTime = 60;
82  bestLoad = inf;
83  bestSol = [];
84  totalCycles = 0;
85  bestCycle = 0;
86  startTime = tic;
87
88  % "correr" o algoritmo Multi Start Hill Climbing
89  while toc(startTime) < maxTime
90      % solu   o inicial
91      [sol, load] = greedyRandomizedStrategy(nNodes, Links, Taux, sP,
            nSP);
92
93      % melhorar a solu   o inicial
94      [sol, load] = HillClimbingStrategy(nNodes, Links, Taux, sP, nSP,
            sol, load);
```

```
95
96      totalCycles = totalCycles + 1;
97
98      % verificar se a solu   o atual    a melhor encontrada
99      if load < bestLoad
100         bestLoad = load;
101         bestSol = sol;
102         bestTime = toc(startTime);
103         bestCycle = totalCycles;
104     end
105 end
106
107 % mostrar os resultados
108 fprintf('Multi Start Hill Climbing algorithm with initial Greedy
        Randomized (Anycast nodes: %d %d)\n', anycastNodes(1),
        anycastNodes(2));
109 fprintf('Worst link load of the network: %.2f\n', bestLoad);
110 fprintf('Total number of cycles run: %d\n', totalCycles);
111 fprintf('Running time at which the best solution was obtained: %.2f
        seconds\n', bestTime);
112 fprintf('Number of cycles at which the best solution was obtained: %d\
        n', bestCycle);
```

Na Task 3.a, o objetivo era usar o algoritmo *Multi Start Hill Climbing* com a estratégia inicial *Greedy Randomized*, usando os nós anycast 3 e 10, para reduzir a pior carga de link na rede.

3.1.2 Resultados e Conclusões

```
------------------------------ Task 3.a.------------------------------
Multi Start Hill Climbing algorithm with initial Greedy Randomized (Anycast nodes: 3 10)
Worst link load of the network: 64.20
Total number of cycles run: 1385
Running time at which the best solution was obtained: 15.34 seconds
Number of cycles at which the best solution was obtained: 575
```

Figura 3.1: Resultados correspondentes ao exercício 3.a.

O algoritmo conseguiu encontrar uma boa solução, com uma pior carga de link de 64.20 Gbps. No entanto, demorou mais tempo (15.34 segundos) e precisou de muitos ciclos para encontrar essa solução (575 ciclos), o que mostra que o processo de busca foi mais longo do que em outras tarefas similares.

3.2 Exercício 3.b.

3.2.1 Código

```
1 %% 3.b.
2
3 clear
4
```

```
fprintf('------------------------------ Task 3.b
    .------------------------------\n');

% carregar os dados
load('InputDataProject2.mat');

% par metros
nNodes = size(Nodes, 1);
nFlows = size(T, 1);
nLinks = size(Links, 1);
k = 12;

v = 2 * 10^5;
D = L / v;

anycastNodes = [1 6];

% inicializar vari veis para os atrasos e caminhos
Taux = zeros(nFlows, 4);
delays = zeros(nFlows, 1);
sP = cell(nFlows, 1);
nSP = zeros(nFlows, 1);
firstPaths = cell(nFlows, 1);
secondPaths = cell(nFlows, 1);

% calcular os caminhos mais curtos e os atrasos de ida e volta
for n = 1:nFlows
    if T(n, 1) == 1
        [shortestPaths, totalCosts] = kShortestPath(D, T(n, 2), T(n,
            3), k);
        sP{n} = shortestPaths;
        nSP(n) = length(shortestPaths);
        delays(n) = totalCosts(1);
        Taux(n, :) = T(n, 2:5);
    elseif T(n, 1) == 2
        [firstPaths{n}, secondPaths{n}, totalPairCosts] =
            kShortestPathPairs(D, T(n, 2), T(n, 3), k);
        sP{n} = firstPaths{n};
        nSP(n) = length(firstPaths{n});
        delays(n) = totalPairCosts(1);
        Taux(n, :) = T(n, 2:5);
    elseif T(n, 1) == 3
        if ismember(T(n, 2), anycastNodes)
            sP{n} = {T(n, 2)};
            nSP(n) = 1;
            Taux(n, :) = T(n, 2:5);
            Taux(n, 3) = T(n, 2);
        else
            cost = inf;
            Taux(n, :) = T(n, 2:5);
            for i = anycastNodes
                [shortestPaths, totalCosts] = kShortestPath(D, T(n, 2)
                    , i, k);
                if totalCosts(1) < cost
                    sP{n} = shortestPaths;
                    nSP(n) = length(shortestPaths);
                    cost = totalCosts(1);
                    delays(n) = totalCosts(1);
                    Taux(n, 3) = i;
                end
            end
        end
```

```
    end
end

unicastFlows1 = find(T(:, 1) == 1);
unicastFlows2 = find(T(:, 1) == 2);
anycastFlows = find(T(:, 1) == 3);

maxDelayUnicast1 = max(delays(unicastFlows1)) * 2 * 1000;
avgDelayUnicast1 = mean(delays(unicastFlows1)) * 2 * 1000;

maxDelayUnicast2 = max(delays(unicastFlows2)) * 2 * 1000;
avgDelayUnicast2 = mean(delays(unicastFlows2)) * 2 * 1000;

maxDelayAnycast = max(delays(anycastFlows)) * 2 * 1000;
avgDelayAnycast = mean(delays(anycastFlows)) * 2 * 1000;

% par metros do algoritmo
maxTime = 60;
bestLoad = inf;
bestSol = [];
totalCycles = 0;
bestCycle = 0;
startTime = tic;

% "correr" o algoritmo Multi Start Hill Climbing
while toc(startTime) < maxTime
    % solu   o inicial
    [sol, load] = greedyRandomizedStrategy(nNodes, Links, Taux, sP,
        nSP);

    % melhorar a solu   o inicial
    [sol, load] = HillClimbingStrategy(nNodes, Links, Taux, sP, nSP,
        sol, load);

    totalCycles = totalCycles + 1;

    % verificar se a solu   o atual    a melhor encontrada
    if load < bestLoad
        bestLoad = load;
        bestSol = sol;
        bestTime = toc(startTime);
        bestCycle = totalCycles;
    end
end

% mostrar os resultados
fprintf('Multi Start Hill Climbing algorithm with initial Greedy
    Randomized (Anycast nodes: %d %d)\n', anycastNodes(1),
    anycastNodes(2));
fprintf('Worst link load of the network: %.2f\n', bestLoad);
fprintf('Total number of cycles run: %d\n', totalCycles);
fprintf('Running time at which the best solution was obtained: %.2f
    seconds\n', bestTime);
fprintf('Number of cycles at which the best solution was obtained: %d\
    n', bestCycle);
```

A Task 3.b, tem o mesmo objetivo da task anterior que é aplicar o algoritmo *Multi Start Hill Climbing* com uma solução inicial baseada na estratégia *Greedy Randomized*, utilizando os nós anycast 1 e 6.

3.2.2 Resultados e Conclusões

```
------------------------------ Task 3.b.------------------------------
Multi Start Hill Climbing algorithm with initial Greedy Randomized (Anycast nodes: 1 6)
Worst link load of the network: 60.20
Total number of cycles run: 2735
Running time at which the best solution was obtained: 0.09 seconds
Number of cycles at which the best solution was obtained: 5
```

Figura 3.2: Resultados correspondentes ao exercício 3.b.

O algoritmo funcionou bem nessa configuração, achando a melhor solução rápido, em 0.09 segundos e no 5º ciclo. A pior carga de link ficou em 60.20 Gbps, o que é bem melhor do que na Task 3.a, que tinha 64.20 Gbps. Também fez muitos ciclos (2735), mostrando que procurou bastante para garantir uma boa solução.

Os nós anycast 1 e 6 foram uma boa escolha, ajudando a distribuir melhor o tráfego e evitar congestionamento nos links mais usados.

3.3 Exercício 3.c.

3.3.1 Código

```
%% 3.c.

clear

fprintf('------------------------------ Task 3.c
    .------------------------------\n');

% carregar os dados
load('InputDataProject2.mat');

% par metros
nNodes = size(Nodes, 1);
nFlows = size(T, 1);
nLinks = size(Links, 1);
k = 12;

v = 2 * 10^5;
D = L / v;

anycastNodes = [4 12];

% inicializar vari veis para os atrasos e caminhos
Taux = zeros(nFlows, 4);
delays = zeros(nFlows, 1);
sP = cell(nFlows, 1);
nSP = zeros(nFlows, 1);
firstPaths = cell(nFlows, 1);
secondPaths = cell(nFlows, 1);

% calcular os caminhos mais curtos e os atrasos de ida e volta
```

```
for n = 1:nFlows
    if T(n, 1) == 1
        [shortestPaths, totalCosts] = kShortestPath(D, T(n, 2), T(n,
            3), k);
        sP{n} = shortestPaths;
        nSP(n) = length(shortestPaths);
        delays(n) = totalCosts(1);
        Taux(n, :) = T(n, 2:5);
    elseif T(n, 1) == 2
        [firstPaths{n}, secondPaths{n}, totalPairCosts] =
            kShortestPathPairs(D, T(n, 2), T(n, 3), k);
        sP{n} = firstPaths{n};
        nSP(n) = length(firstPaths{n});
        delays(n) = totalPairCosts(1);
        Taux(n, :) = T(n, 2:5);
    elseif T(n, 1) == 3
        if ismember(T(n, 2), anycastNodes)
            sP{n} = {T(n, 2)};
            nSP(n) = 1;
            Taux(n, :) = T(n, 2:5);
            Taux(n, 3) = T(n, 2);
        else
            cost = inf;
            Taux(n, :) = T(n, 2:5);
            for i = anycastNodes
                [shortestPaths, totalCosts] = kShortestPath(D, T(n, 2)
                    , i, k);
                if totalCosts(1) < cost
                    sP{n} = shortestPaths;
                    nSP(n) = length(shortestPaths);
                    cost = totalCosts(1);
                    delays(n) = totalCosts(1);
                    Taux(n, 3) = i;
                end
            end
        end
    end
end

unicastFlows1 = find(T(:, 1) == 1);
unicastFlows2 = find(T(:, 1) == 2);
anycastFlows = find(T(:, 1) == 3);

maxDelayUnicast1 = max(delays(unicastFlows1)) * 2 * 1000;
avgDelayUnicast1 = mean(delays(unicastFlows1)) * 2 * 1000;

maxDelayUnicast2 = max(delays(unicastFlows2)) * 2 * 1000;
avgDelayUnicast2 = mean(delays(unicastFlows2)) * 2 * 1000;

maxDelayAnycast = max(delays(anycastFlows)) * 2 * 1000;
avgDelayAnycast = mean(delays(anycastFlows)) * 2 * 1000;

% par metros do algoritmo
maxTime = 60;
bestLoad = inf;
bestSol = [];
totalCycles = 0;
bestCycle = 0;
startTime = tic;

% "correr" o algoritmo Multi Start Hill Climbing
while toc(startTime) < maxTime
```

```
89      % solu   o inicial
90      [sol, load] = greedyRandomizedStrategy(nNodes, Links, Taux, sP,
            nSP);
91
92      % melhorar a solu   o inicial
93      [sol, load] = HillClimbingStrategy(nNodes, Links, Taux, sP, nSP,
            sol, load);
94
95      totalCycles = totalCycles + 1;
96
97      % verificar se a solu   o atual    a melhor encontrada
98      if load < bestLoad
99          bestLoad = load;
100         bestSol = sol;
101         bestTime = toc(startTime);
102         bestCycle = totalCycles;
103     end
104 end
105
106 % mostrar os resultados
107 fprintf('Multi Start Hill Climbing algorithm with initial Greedy
        Randomized (Anycast nodes: %d %d)\n', anycastNodes(1),
        anycastNodes(2));
108 fprintf('Worst link load of the network: %.2f\n', bestLoad);
109 fprintf('Total number of cycles run: %d\n', totalCycles);
110 fprintf('Running time at which the best solution was obtained: %.2f
        seconds\n', bestTime);
111 fprintf('Number of cycles at which the best solution was obtained: %d\
        n', bestCycle);
```

A Task 3.c tem como objetivo aplicar o algoritmo Multi Start Hill Climbing com uma solução inicial baseada na estratégia Greedy Randomized, utilizando os nós anycast 4 e 12.

3.3.2 Resultados e Conclusões

```
------------------------------ Task 3.c.------------------------------
Multi Start Hill Climbing algorithm with initial Greedy Randomized (Anycast nodes: 4 12)
Worst link load of the network: 59.70
Total number of cycles run: 2286
Running time at which the best solution was obtained: 4.00 seconds
Number of cycles at which the best solution was obtained: 144
```

Figura 3.3: Resultados correspondentes ao exercício 3.c.

O algoritmo, relativamente aos resultados anteriores, teve um desempenho muito bom nesta configuração, conseguindo reduzir a pior carga de link para 59.70 Gbps. A melhor solução foi encontrada em 4.00 segundos e no 144º ciclo, mostrando que o processo de otimização funcionou bem para essa configuração.

A escolha dos nós anycast 4 e 12 se mostrou eficiente, ajudando a distribuir o tráfego de maneira equilibrada e evitando sobrecarga nos links mais utilizados.

3.4 Exercício 3.d.

3.4.1 Código

```
%% 3.d.

clear

fprintf('------------------------------ Task 3.d
    .------------------------------\n');

% carregar os dados
load('InputDataProject2.mat');

% par metros
nNodes = size(Nodes, 1);
nFlows = size(T, 1);
nLinks = size(Links, 1);
k = 12;

v = 2 * 10^5;
D = L / v;

anycastNodes = [5 14];

% inicializar vari veis para os atrasos e caminhos
Taux = zeros(nFlows, 4);
delays = zeros(nFlows, 1);
sP = cell(nFlows, 1);
nSP = zeros(nFlows, 1);
firstPaths = cell(nFlows, 1);
secondPaths = cell(nFlows, 1);

% calcular os caminhos mais curtos e os atrasos de ida e volta
for n = 1:nFlows
    if T(n, 1) == 1
        [shortestPaths, totalCosts] = kShortestPath(D, T(n, 2), T(n,
            3), k);
        sP{n} = shortestPaths;
        nSP(n) = length(shortestPaths);
        delays(n) = totalCosts(1);
        Taux(n, :) = T(n, 2:5);
    elseif T(n, 1) == 2
        [firstPaths{n}, secondPaths{n}, totalPairCosts] =
            kShortestPathPairs(D, T(n, 2), T(n, 3), k);
        sP{n} = firstPaths{n};
        nSP(n) = length(firstPaths{n});
        delays(n) = totalPairCosts(1);
        Taux(n, :) = T(n, 2:5);
    elseif T(n, 1) == 3
        if ismember(T(n, 2), anycastNodes)
            sP{n} = {T(n, 2)};
            nSP(n) = 1;
            Taux(n, :) = T(n, 2:5);
            Taux(n, 3) = T(n, 2);
        else
            cost = inf;
            Taux(n, :) = T(n, 2:5);
            for i = anycastNodes
                [shortestPaths, totalCosts] = kShortestPath(D, T(n, 2)
                    , i, k);
```

```
                if totalCosts(1) < cost
                    sP{n} = shortestPaths;
                    nSP(n) = length(shortestPaths);
                    cost = totalCosts(1);
                    delays(n) = totalCosts(1);
                    Taux(n, 3) = i;
                end
            end
        end
    end
end

unicastFlows1 = find(T(:, 1) == 1);
unicastFlows2 = find(T(:, 1) == 2);
anycastFlows = find(T(:, 1) == 3);

maxDelayUnicast1 = max(delays(unicastFlows1)) * 2 * 1000;
avgDelayUnicast1 = mean(delays(unicastFlows1)) * 2 * 1000;

maxDelayUnicast2 = max(delays(unicastFlows2)) * 2 * 1000;
avgDelayUnicast2 = mean(delays(unicastFlows2)) * 2 * 1000;

maxDelayAnycast = max(delays(anycastFlows)) * 2 * 1000;
avgDelayAnycast = mean(delays(anycastFlows)) * 2 * 1000;

% par metros do algoritmo
maxTime = 60;
bestLoad = inf;
bestSol = [];
totalCycles = 0;
bestCycle = 0;
startTime = tic;

% "correr" o algoritmo Multi Start Hill Climbing
while toc(startTime) < maxTime
    % solu   o inicial
    [sol, load] = greedyRandomizedStrategy(nNodes, Links, Taux, sP,
        nSP);

    % melhorar a solu   o inicial
    [sol, load] = HillClimbingStrategy(nNodes, Links, Taux, sP, nSP,
        sol, load);

    totalCycles = totalCycles + 1;

    % verificar se a solu   o atual    a melhor encontrada
    if load < bestLoad
        bestLoad = load;
        bestSol = sol;
        bestTime = toc(startTime);
        bestCycle = totalCycles;
    end
end

% mostrar os resultados
fprintf('Multi Start Hill Climbing algorithm with initial Greedy
    Randomized (Anycast nodes: %d %d)\n', anycastNodes(1),
    anycastNodes(2));
fprintf('Worst link load of the network: %.2f\n', bestLoad);
fprintf('Total number of cycles run: %d\n', totalCycles);
fprintf('Running time at which the best solution was obtained: %.2f
    seconds\n', bestTime);
```

```
111 fprintf('Number of cycles at which the best solution was obtained: %d\
    n', bestCycle);
```

A Task 3.d tem também como objetivo aplicar o algoritmo *Multi Start Hill Climbing* com uma solução inicial baseada na estratégia *Greedy Randomized*, utilizando os nós anycast 5 e 14.

3.4.2 Resultados e Conclusões

```
------------------------------ Task 3.d.------------------------------
Multi Start Hill Climbing algorithm with initial Greedy Randomized (Anycast nodes: 5 14)
Worst link load of the network: 62.90
Total number of cycles run: 3350
Running time at which the best solution was obtained: 0.08 seconds
Number of cycles at which the best solution was obtained: 3
```

Figura 3.4: Resultados correspondentes ao exercício 3.d.

O algoritmo conseguiu encontrar a melhor solução de forma muito rápida, em apenas 0.08 segundos e no 3º ciclo. No entanto, a pior carga de link registada foi de 62.90 Gbps, o que mostra que a configuração inicial com os nós anycast 5 e 14 não foi tão eficiente para redistribuir o tráfego e reduzir os congestionamentos.

Mesmo assim, o número total de ciclos executados (3350) indica que o algoritmo continuou a explorar outras soluções, garantindo que não havia alternativas melhores disponíveis. Embora os nós anycast escolhidos não tenham sido os mais eficazes para minimizar a carga de link, o processo de otimização mostrou-se rápido e funcional.

3.5 Exercício 3.e.

3.5.1 Comparação das soluções obtidas na task 2 e 3.

Worst Link Load:

- Na Task 2, a pior carga de link variou entre 60.00 Gbps (Task 2.c) e 64.20 Gbps (Task 2.a).
- Na Task 3, os resultados variaram entre 59.70 Gbps (Task 3.c) e 64.20 Gbps (Task 3.a), mostrando que a Task 3 conseguiu uma ligeira melhoria nos melhores casos.

A melhoria entre as duas tasks é atribuída ao uso da função *kShortestPathPairs*, que permite encontrar pares de caminhos disjuntos e distribuir melhor o tráfego pela rede.
Tempo para encontrar a melhor solução:

- A Task 2 teve tempos significativamente menores em geral, com soluções ótimas encontradas em menos de 0.5 segundos na maioria das configurações.
- A Task 3 teve tempos mais variáveis, indo desde 0.08 segundos (Task 3.d) até 15.34 segundos (Task 3.a).

Escolha dos Nós Anycast:

- Na Task 2 e na Task 3, os nós anycast 4 e 12 foram os melhores, com a menor carga de link em ambas as tasks (60.00 Gbps na Task 2.c e 59.70 Gbps na Task 3.c). Essa pequena melhoria na Task 3 aconteceu porque o uso da função kShortestPathPairs ajudou a dividir o tráfego de forma mais equilibrada, escolhendo pares de caminhos e evitando que alguns links ficassem muito cheios.

Número Total de Ciclos Executados:

- A Task 3 precisou de mais ciclos em algumas configurações, como 3350 ciclos na Task 3.d contra 2494 ciclos na Task 2.c. Isso aconteceu porque *kShortestPathPairs* aumenta as opções de caminhos e o algoritmo leva mais tempo a processar.

3.5.2 Diferenças no desempenho do algoritmo

O algoritmo *Multi Start Hill Climbing* funcionou bem nas duas tasks, mas o uso da função *kShortestPathPairs* na Task 3 trouxe algumas vantagens:

- Ao considerar pares de caminhos disjuntos, a Task 3 conseguiu distribuir o tráfego de forma mais equilibrada, o que ajudou a reduzir ligeiramente a carga máxima de link.
- Apesar das melhorias nos resultados, a Task 3 precisou de mais ciclos e, em alguns casos, levou mais tempo para encontrar as melhores soluções, o que era esperado devido à maior quantidade de opções de caminhos analisadas.

Autoavaliação

Neste projeto, o esforço foi distribuído igualmente entre os membros do grupo. Cada um contribuiu com 50% do trabalho, colaborando em todas as fases do projeto, desde a implementação das simulações até a análise dos resultados e elaboração do relatório.

- João Gaspar: 50% do trabalho
- João Monteiro: 50% do trabalho