Universidade de Aveiro

Modelação e Desempenho de Redes e Serviços
Mini-Projeto 2

João Gameiro (93097), Pedro Abreu (93240)

Departamento de Electrónica, Telecomunicações e Informática

3 de fevereiro de 2022

Conteúdo

Lista de Figuras

Capítulo 1

Introdução

O presente relatório visa descrever as decisões e conclusões tiradas durante a resolução do Mini-Projecto 2 desenvolvido no âmbito da unidade curricular de Modelação e Desempenho de Redes e Serviços. Este documento está dividido em 4 capítulos, um por cada tarefa.

É importante referir que no código desenvolvido apresentado neste relatório encontram-se comentários que ajudam numa melhor compreensão do mesmo.

1.1 Dados

Todos o código apresentado neste trabalho tem por base este ficheiro com os dados dos exercícios e definição de variáveis que são usadas nas várias tarefas. Destacar que se calcula número de nós, nNodes, número de *links*, nLinks, o número de *flows*, nFlows, o matrix comprimentos dos *links*, L, a matrix do tempo médio entre falhas dos *links*, T, matrix que contém os fluxos, MTBF, e a matrix *availability* entre *links*, A e logA, matrix para uniformizar o inverso da matrix A, ou seja quanto maior a availability menor o custo.

De notar que nem todos as tarefas apresentadas usam todas as variáveis anteriormente referidas.

```
Nodes= [30 70
        350 40
        550 180
        310 130
        100 170
        540 290
        120 240
        400 310
        220 370
        550 380];

Links= [1 2
         1 5
         2 3
```

```
          2 4
          3 4
          3 6
          3 8
          4 5
          4 8
          5 7
          6 8
          6 10
          7 8
          7 9
          8 9
          9 10];

T= [1  3  1.0 1.0
    1  4  0.7 0.5
    2  7  2.4 1.5
    3  4  2.4 2.1
    4  9  1.0 2.2
    5  6  1.2 1.5
    5  8  2.1 2.5
    5  9  1.6 1.9
    6 10  1.4 1.6];

nNodes= 10;
nLinks= size(Links,1);
nFlows= size(T,1);

B= 625;  %Average packet size in Bytes

co= Nodes(:,1)+1i*Nodes(:,2);

L= inf(nNodes);    %Square matrix with arc lengths (in Km)
for i=1:nNodes
    L(i,i)= 0;
end
C= zeros(nNodes);  %Square matrix with arc capacities (in Gbps)
for i=1:nLinks
    C(Links(i,1),Links(i,2))= 10;  %Gbps
    C(Links(i,2),Links(i,1))= 10;  %Gbps
    d= abs(co(Links(i,1))-co(Links(i,2)));
    L(Links(i,1),Links(i,2))= d+5; %Km
    L(Links(i,2),Links(i,1))= d+5; %Km
end
L= round(L);  %Km

MTBF= (450*365*24)./L;
A= MTBF./(MTBF + 24);
% matrix A, matrix com a disponibilidade de cada ligacao
A(isnan(A))= 0;

% links com valor mais perto de 0 sao os mais curtos
logA = -log(A);
```

Capítulo 2

Tarefa 1

2.1 alínea A

Código

Nesta alínea era pretendido que se calculasse o número de diferentes caminhos de *routing* para cada fluxo. Para isso foi criada uma função auxiliar que recebia como argumentos o número de caminhos a calcular, a matriz L que representa a distância dos *links* e a matriz T que contém os fluxos. Esta função auxiliar por sua vez chama a função *kShortestPath* para calcular n *shortest paths* e a soma dos seus respectivos custos.

```
% 1.a)
% Compute up to 100 paths for each flow:
n= inf;
[sP, nSP]= calculatePaths(L,T,n);

fprintf('Number of diferent paths for each flow: ');
nSP
```

Função auxiliar *calculatePaths*.

```
function [sP nSP]= calculatePaths(L,T,n)
    nFlows= size(T,1);
    nSP= zeros(1,nFlows);
    for i=1:nFlows
        [shortestPath, totalCost] = kShortestPath(L,T(i,1),T(i,2),n);
        sP{i}= shortestPath;
        nSP(i)= length(totalCost);
    end
end
```

Resultados

Como podemos ver foi obtido um vector com 9 números. Em cada posição do vector está o número de diferentes caminhos calculados para aquele fluxo. O fluxo com o menor número de caminhos diferentes é o 4 e o que tem mais caminhos diferentes por sua vez é o 8.

```
Number of diferent paths for each flow:
nSP =

    32    32    38    24    36    37    25    41    28
```

Figura 2.1: Resultado da alínea 1.a).

2.2 alínea B

Código

O objectivo desta alínea era criar e correr um algoritmo que computasse uma solução de *routing* simétrica de um único caminho para cada fluxo da rede de modo a suportar um serviço que minimiza-se a pior *link load*.

O algoritmo desenvolvido, consiste em durante 10 segundos, repetidamente escolher aleatoriamente um *shortest path* para cada fluxo e a partir de todos os resultados obtidos escolher aquele que providencia uma menor *link load*. O algoritmo foi corrido para três situações diferentes: considerando todos os caminhos de *routing*, ou apenas os 10 menores caminhos ou apenas os 5 menores caminhos. De notar que são os caminhos com menor número de *hops*.

Para poder suportar todas estes alternativas foi criado um vetor com os valores [*inf, 10, 5*] e para cada um dos valores do vetor, quando fosse feita a escolha aleatória, era tido em conta o número de menores caminhos de entre os quais escolher.

```
%% 1.b) % random strat
%Values of all routing paths, 10 shortest paths and 5 shortest paths
limits = [inf, 10, 5];

fprintf('RANDOM:\n');
figure('Name','Ex. 1.b)','NumberTitle','off');

for limit = limits
    t= tic;
    bestLoad= inf;
    sol= zeros(1,nFlows);
    allValues= [];
    while toc(t)<10
        for i= 1:nFlows
            % choose n shortest path for each flow
            % the choise is made between limit and all k-shortest paths
            n = min(limit,nSP(i));
```

```
            sol(i)= randi(n); %randomly choose one of the shortest paths
        end
        %Calculate the load for the obtained solution
        Loads= calculateLinkLoads(nNodes,Links,T,sP,sol);
        load= max(max(Loads(:,3:4)));
        allValues= [allValues load]; %Store value for plotting
        if load<bestLoad    %Update best Load if necessary
            bestSol= sol;
            bestLoad= load;
        end
    end
    ...
    % prints and graphic
    ...
end
...
% prints and graphic
...
```

Resultados

Analisando os resultados obtidos podemos comprovar à medida que se reduz o número de caminhos de entre os quais se vai escolher a solução, melhores resultados obtemos (menor valor de *worst load* e média dos mesmos).

```
RANDOM:
Inf best paths
   Best load = 4.90 Gbps
   No. of solutions = 133850
   Averg. quality of solutions = 10.32 Gbps
10 best paths
   Best load = 4.40 Gbps
   No. of solutions = 135763
   Averg. quality of solutions = 8.51 Gbps
5 best paths
   Best load = 4.00 Gbps
   No. of solutions = 135460
   Averg. quality of solutions = 7.74 Gbps
```

Figura 2.2: Resultado da alínea 1.b).

O gráfico (Figura 2.3) de todas as soluções obtidas também nos confirma que a qualidade das soluções aumenta com a diminuição do número de *paths* a escolher. Para perceber porque é que isto acontece é necessário entender como o algoritmo escolhe a solução e a resposta a isso é de forma aleatória. Para cada fluxo, dado um certo número de caminhos, o algoritmo escolhe um de forma aleatória e verifica se a solução com essa caminho é menor que a anterior ou não. Logo ao considerarmos um maior número de caminhos estamos a considerar um valor

muito mais elevado de soluções de entre as quais iremos escolher um valor aleatoriamente. Tal facto aumenta a probabilidade de escolhermos uma pior solução. Por sua vez escolhendo de entre apenas 5 caminhos mais curtos o espaço de escolha é muito menor, o que reduz a probabilidade da escolha de um caminho que prejudique substancialmente a carga da rede e dê uma pior solução.

Concluindo, quando aumentamos o número de caminhos de entre os quais vamos escolher, estamos simplesmente a aumentar o espaço de procura de soluções para considerar um número maior de soluções (mesmo que isso inclua um decréscimo na qualidade das mesmas). E como o algoritmo escolhe sempre uma solução aleatória não podemos garantir que das escolhas de solução sejam excluídas opções irrelevantes (que dão pior resultado de *link load*).

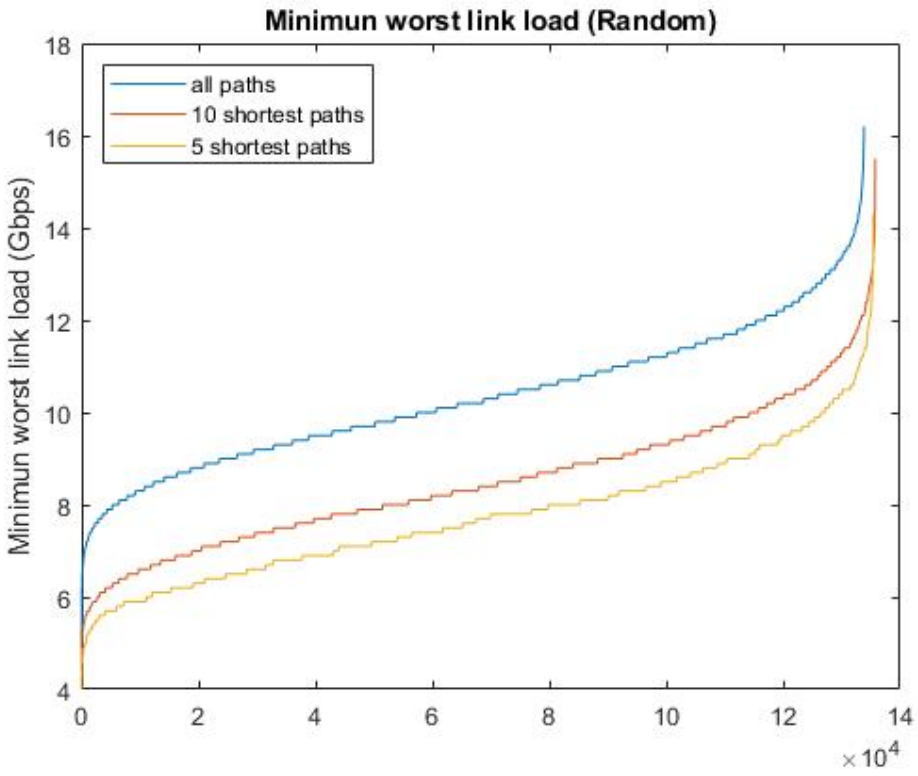

Figura 2.3: Gráfico resultante da execução da alínea 1.b).

Em termos do número de soluções encontradas é praticamente semelhante em todos os casos, pelo facto do processo de escolha ser o mesmo em todos os casos, mudando apenas o número de opções de entre os quais escolher. Ao correr o algoritmo várias vezes podemos obter oscilações do número de soluções obtidas mas no geral ficam todos bastante perto uns dos outros.

2.3 alínea C

Código

O objetivo deste programa era o mesmo da alínea anterior mas desta vez com a estratégia *Greedy Randomized*. Esta estratégia difere da anterior na medida em que a aleatoriedade da escolha dos caminhos é ligeiramente distinta. Em vez de ser escolhido para cada *flow* um caminho aleatoriamente, os fluxos são ordenados de forma aleatória, e para cada fluxo seleccionamos o caminho que oferece a menor *worst load*. Tal como na alínea anterior o algoritmo corre num ciclo *while* com uma duração de 10 segundos.

```
%% 1.c) % greedy randomized strat
%Values of all routing paths, 10 shortest paths and 5 shortest paths
limits = [inf, 10, 5];
fprintf('GREEDY RANDOMIZED:\n');
figure('Name','Ex. 1.c)','NumberTitle','off');
for limit = limits
    t= tic;
    bestLoad= inf;
    allValues= [];
    while toc(t)<10
        ax2 = randperm(nFlows); % random order of flows
        sol= zeros(1,nFlows);   % vector for solution
        for i= ax2
            k_best = 0;
            best = inf;
            n = min(limit, nSP(i));
            for k = 1:n
                sol(i)= k;
                Loads= calculateLinkLoads(nNodes,Links,T,sP,sol);
                load= max(max(Loads(:,3:4)));
                if load < best
                    k_best = k;
                    best = load;
                end
            end
            sol(i) = k_best;
        end
        load= best;
        allValues= [allValues load];
        if load<bestLoad
            bestSol= sol;
            bestLoad= load;
        end
    end
    ...
    % prints and graphic
    ...
end
...
% prints and graphic
...
```

Resultados

Analisando os resultados obtidos podemos concluir que à medida que é reduzido o número de caminhos de *routing* de entre os quais vai ser escolhido a solução, que a qualidade média das soluções piora. O mesmo se verifica no valor para *worst load* encontrado que é pior no caso em que o número de routing paths é menor. Analisando o código, concluímos que o algoritmo *greedy randomized* irá sempre dar prioridade de escolha aos caminhos que apresentam menor *link load*. Logo ao aumentar o espaço de soluções não estamos necessariamente a introduzir piores soluções pelo facto de o algoritmo escolher sempre a melhor solução. Aumentar o espaço de soluções implica melhorar a precisão de escolha do algoritmo. Com apenas os 5 menores caminhos podemos concluir que a combinação dos 5 menores caminhos para cada fluxo não é necessariamente a melhor. Neste exercícios, o melhor resultado obtido é quando escolhemos uma combinação entre os 10 menores caminhos de cada fluxo. A combinação de todos os caminhos do fluxo também obtém bons resultados, no entanto calcula muito menos soluções.

Com apenas 5 caminhos para escolher por cada fluxo podemos deixar soluções importantes de fora, por sua vez, escolhendo de todos os caminhos temos em mãos todos os possíveis caminhos e garantimos que o algoritmo irá escolher a melhor solução não deixando nenhuma possibilidade importante de fora. Naturalmente é compreensível que quanto menor for o número de caminhos a escolher, pior poderá ser a solução (poderá porque nada impede que a melhor solução esteja por acaso nesses 5 *shortest paths*). É importante referir que o algoritmo *Greedy* pode não encontrar a solução ideal apenas em 10 segundos (visto que é computacionalmente muito exigente).

```
GREEDY RANDOMIZED:
Inf best paths
   Best load = 3.70 Gbps
   No. of solutions = 4003
   Averg. quality of solutions = 4.54 Gbps
10 best paths
   Best load = 3.70 Gbps
   No. of solutions = 13260
   Averg. quality of solutions = 4.52 Gbps
5 best paths
   Best load = 4.00 Gbps
   No. of solutions = 25220
   Averg. quality of solutions = 5.06 Gbps
```

Figura 2.4: Resultado da alínea 1.c).

Tendo em conta que o algoritmo *greedy randomized* percorre, para cada fluxo todos as possibilidades e guarda aquele que der o menor valor de *worst link load*, então quando mais caminhos tiver em cada fluxo, mais iterações terá de fazer. Sendo que tem de fazer mais iterações, então, menos soluções encontra devido ao facto de perder mais tempo a procurar cada solução. Logo isso justifica os valores dos gráficos. Com 5 *shortest paths*, em cada

iteração o algoritmo escolhe apenas de entre 5 caminhos, o que tendo em conta que temos 9 fluxos, implica fazer 45 iterações e dá origem muitas soluções. Para 10 *shortest paths* já temos de fazer 90 iterações o que reduz o bastante o número de soluções. Considerando todos os caminhos o número de iterações é muito maior o que nos deixa muito menos tempo para procurar outras soluções logo esta é a alternativa que apresenta menos soluções.

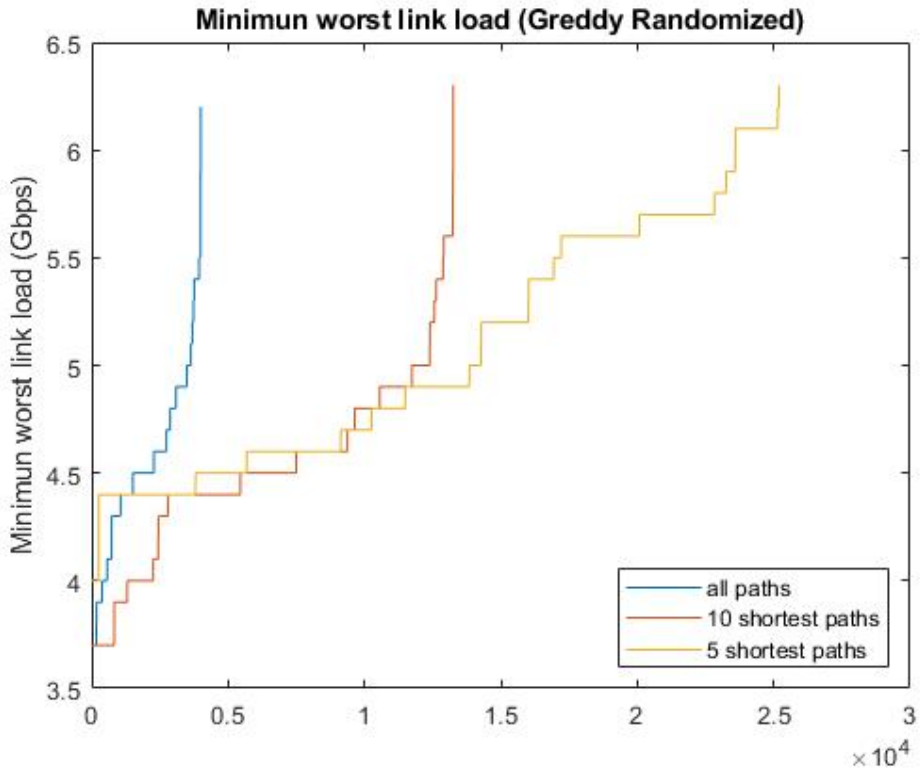

Figura 2.5: Gráfico resultante da execução da alínea 1.c).

2.4 alínea D

Código

Para este caso o que se pretendia era o mesmo que nas duas alíneas anteriores, mas desta vez com o algoritmo *Multi Start Hill Climbing*.

O algoritmo *Multi Start Hill Climbing* é a combinação de dois algoritmos. O algoritmo *Greedy Randomized* e o algoritmo de *Hill Climbing*. Através da estratégia *Greedy Randomized*, conseguimos calcular uma solução. Por sua vez, tendo obtido a solução, são calculadas as soluções vizinhas e é seleccionada a melhor. Se essa solução for melhor que a actual então repetimos os passos de cálculo de soluções vizinhas. Caso contrário encontrámos a solução ideal. O que foi descrito anteriormente não representa só o que é o *Multi Start Hill Climbing*. Para completarmos este algoritmo falta apenas repetir todos os passos referidos um certo

número de vezes (indicando um critério de paragem). Resumindo constrói-se várias vezes uma solução de raiz e tentamos melhorá-la o máximo possível,

Neste caso esse critério de paragem é o tempo, ou seja o algoritmo repete-se durante um total de 10 segundos. Tendo obtido uma solução a partir do algoritmo *Greedy Randomized*, calculamos para cada fluxo soluções vizinhas e guardamos a melhor delas. Finalmente verificamos se a solução vizinha obtida é melhor que a actual e se for voltamos a calcular os vizinhos. Se não for significa que a solução local encontrada é a ideal. Finalmente (e como o algoritmo corre várias vezes durante 10 segundos), é preciso verificar se a solução local encontrada é melhor que a global. Se for melhor são actualizados os valores de melhor solução global e melhor *Load* global.

```
%% 1.d) % hill climbing multi start
fprintf('MULTI START HILL CLIMBING:\n');
figure('Name','Ex. 1.d)','NumberTitle','off');
limits = [inf, 10, 5];
for limit = limits
    %Build a multi start hill climbing solution
    globalBestLoad= inf;
    allValues= [];
    t=tic;
    while toc(t) < 10
        %Greedy Randomized Solution
        [bestSol,bestLoad] = greedyRandomizedLoads(nFlows,nSP, nNodes, ...
            Links, T, sP, limit);
        repeat = true;
        while repeat
            %Iterate through all values of the solution (to calculate best ...
                neighbor)
            neighborBest= inf;
            for i=1:nFlows
                [nS, nL]= BuildNeighbor(bestSol,i, sP, nSP, Links, nNodes, ...
                    T, bestLoad);
                if nL < neighborBest
                    neighborBest= nL;
                    neighborSol= nS;
                end
            end
            if neighborBest < bestLoad
                bestSol= neighborSol;
                bestLoad= neighborBest;
            else
                repeat= false;
            end
        end
        allValues= [allValues bestLoad];
        if bestLoad < globalBestLoad
            globalBestLoad= bestLoad;
            globalSol= bestSol;
        end
    end
...
% prints and graphic
```

```
...
end
...
% prints and graphic
...
```

De modo a tornar o código do algoritmo *Multi Start Hill Climbing* mais legível foram criadas duas funções auxiliares. Uma delas tem por base o código da alínea anterior e serve para calcular uma solução e a *load* dessa solução usando o algoritmo *Greedy Randomized*.

```
function [sol,load] = greedyRandomizedLoads(nFlows,nSP, nNodes, Links, T, ...
    sP, limit)
ax2 = randperm(nFlows); % random order
sol= zeros(1,nFlows);
for i= ax2
    k_best = 0;
    best = inf;
    n = min(limit, nSP(i));
    for k= 1:n
        sol(i)= k;
        Loads= calculateLinkLoads(nNodes,Links,T,sP,sol);
        load= max(max(Loads(:,3:4)));
        if load < best
            k_best = k;
            best = load;
        end
    end
    sol(i) = k_best;
end
load= best;
end
```

A segunda função criada foi a *BuildNeighbor* que serve, para um dado fluxo de uma certa solução, calcular todas as soluções vizinhas e devolver a melhor. Esta função basicamente itera sobre todos os fluxos e para cada fluxo retira os seus vizinhos. Tendo um vizinho é calculada a sua *load*. Se a *load* calculada for melhor que a actual então guardamos em variáveis auxiliares informação para que no fim se possa devolver essa solução.

```
function [nSol, nL] = BuildNeighbor(bestSol,i, sP, nSP, Links, nNodes, T, ...
    nLoad)
%Find best neighbor of a given solution
%   i - flow from whom we will choose neighbors
%   bestSol - current solution
%   sP - Contains shortest paths among network nodes for each flow
%   nSP - Contains the number of shortest paths for each flow
%   Links - Links of the network
%   nNodes - Network Nodes
%   T - Network flows
%   nLoad - best load found with  greedy randomized solution

nFlows= size(bestSol);
nSol= bestSol;
```

```
nL= nLoad;

k=0;
value=0;

for n=1:nFlows
    if n≠i
        newNeighbor= bestSol;
        for j=1:nSP(n)
            if j≠bestSol(n)
                newNeighbor(n)= j;
                nLoads= calculateLinkLoads(nNodes,Links,T,sP,newNeighbor);
                load= max(max(nLoads(:,3:4)));
                if load < nL
                    k=n;
                    value= j;
                    nL= load;
                end
            end
        end
    end
end
if k>0
    nSol(k)= value;
end

end
```

Resultados

Analisando os resultados obtidos verificamos que a qualidade da *Best Load* decresce com o número de caminhos de entre os quais se escolhe a solução. No entanto no caso da qualidade média das soluções os valores já oscilam ligeiramente, sendo que ao correr várias vezes os algoritmo podemos obter resultados diferentes mas no geral, com 10 *shortest paths* obtemos a melhor qualidade. No caso dos 5 *shortest paths* obtemos sempre a pior qualidade de soluções. Tal como nos casos anteriores, partimos do princípio que ao aumentar o número de caminhos de entres os quais escolhemos, estamos a aumentar o espaço de possíveis soluções. E tal como vimos na alínea anterior, o algoritmo *greedy randomized* apresenta piores resultados para um menor número de caminhos.

Como o *Multi Start Hill Climbing* procura por soluções vizinhas dadas pelo algoritmo *Greedy* então é espectável que a qualidade das soluções fornecidas pelo *Greedy Randomized* influencie também a do *Multi Start*. Isto é, se a solução fornecida pelo *Greedy* não for a melhor então a procura pelos vizinhos pode melhorar ligeiramente a solução mas não nos garante que encontramos a melhor. Logo é compreensivel que quanto melhor a solução fornecida pelo algoritmo *Greedy Randomized*, melhor será a encontrada pelo *Multi Start*, pelo facto de procurar nas "vizinhanças"da mesma solução.

Isto justifica o facto de a qualidade das solução ser pior no caso dos 5 *shortest paths* do que nos outros dois, porque o algoritmo *Greedy* tinha mais dificuldade em encontrar uma boa solução para valores baixos do número de caminhos.

Ao relembrar-mo-nos que o algoritmo *Greedy* fornecia os melhores resultados quando considerava todos os caminhos, salta à vista a questão do porque é que o *Multi Start* tem então piores resultados para todos do que no caso de 10 caminhos. Isto acontece porque a procura nas vizinhanças mai melhorar ligeiramente a solução, logo quando o algoritmo *Greedy* já oferece uma solução boa, a procura de vizinhos melhora essa solução. Logo como o algoritmo *Greedy* demora muito tempo a computar uma solução quando tem de escolher de entre todos os caminhos, o número de iterações pode não ser suficiente para que seja fornecida uma solução com qualidade de modo a que seja encontrado o melhor vizinho possível. Isto é melhores soluções podem ter ficado por visitar.

Com 10 caminhos o algoritmo *Greedy* é mais rápido que permite iterar sobre mais soluções e correr mais vezes a procura de vizinhos, encontrando assim uma melhor solução.

```
MULTI START HILL CLIMBING:
Inf best paths
   Best load = 3.70 Gbps
   No. of solutions = 2545
   Averg. quality of solutions = 4.54 Gbps
10 best paths
   Best load = 3.70 Gbps
   No. of solutions = 3813
   Averg. quality of solutions = 4.51 Gbps
5 best paths
   Best load = 4.00 Gbps
   No. of solutions = 4175
   Averg. quality of solutions = 5.06 Gbps
```

Figura 2.6: Resultado da alínea 1.d).

O número de soluções encontradas volta novamente a estar dependente da rapidez com que o algoritmo *Greedy* consegue computar uma solução. Se demorar bastante tempo ficamos com menos tempo para a procura de vizinhos, o que significa que consequentemente iremos ter um número menor de soluções. Se o *Greedy* conseguir encontrar soluções muito rapidamente ficamos com mais tempo livre para iterar e procurar outras soluções de modo a obter um número maior de soluções. O *Greedy* demora mais tempo quando tem de escolher de entre todos os caminhos e menos tempo quando está reduzido a apenas 10 ou 5. Isso também se reflecte no caso do *Multi-Start Hill Climbing*.

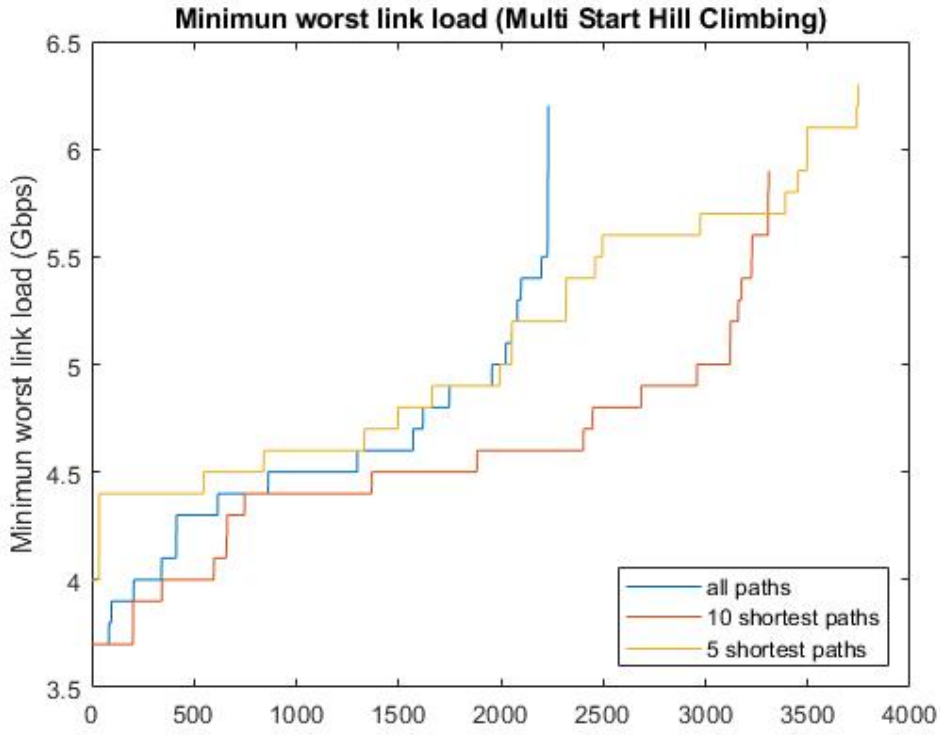

Figura 2.7: Gráfico resultante da execução da alínea 1.d).

2.5 alínea E

Comparando os três algoritmos uma conclusão salta logo à vista, tanto o *Multi Start* como o *Greedy Randomized* apresentam melhores resultados que o algoritmo *Random*. No geral a qualidade das soluções é melhor do que no *Random* assim como a própria solução encontrada. Isto acontece pelo facto de o algoritmo *Random* escolher de forma aleatória a sua solução e se for melhor que a atual (caso esta exista), então passa a considerar essa como sendo a melhor solução. É uma estratégia pouco eficiente pois ao aumentar o espaço de resultados não se pode garantir a qualidade das soluções porque tanto pode acontecer que este algoritmo escolha a melhor que existe como a pior. Por sua vez o algoritmo *Greedy Randomized* apenas vai considerar as soluções que forem as melhores possíveis no momento de escolha. Tal facto faz com que a qualidade geral das soluções seja muito melhor que a do caso *Random*. O algoritmo *Multi Start* por sua vez considera uma solução obtida pelo *Greedy* e tenta melhorá-la procurando soluções vizinhas logo consegue facilmente obter bons resultados sem considerar soluções erradas.

Em termos de *performance*, o algoritmo *Greedy Randomized* é computacionalmente pesado, pois para cada fluxo faz uma pesquisa da solução que satisfaz no seu caso o critério à escolha. O *Random* por sua vez faz apenas uma escolha aleatória da solução de entre um conjunto de caminhos, logo é computacionalmente muito mais leve que o *Greedy*. O *Multi*

Start por sua vez engloba o *Greedy*, e obtem resultados semelhantes ao *Greedy*, não efectuando pesquisas tão intensivas por todas as possibilidades mas procurando apenas pelas melhores soluções vizinhas. No entanto implica um processo de construção de soluções vizinhas que é bastante intensivo, ao acrescento do algoritmo *Greedy*. Estes factos reflectem-se no número de soluções encontradas, o *Multi - Start* consegue encontrar menos soluções mas de qualidade no entanto exige um esforço computacional elevado. Por sua vez o algoritmo *Greedy* percorre um número muito elevado de possíveis soluções, e atinge os mesmos resultados que o *Multi Start*, ou seja o *Multi Start* no geral não foi capaz de melhorar as soluções encontradas pelo *Greedy*. O *Random* por sua vez é muito pouco intensivo mas a qualidade das soluções que apresenta é mais reduzida em relação aos outros dois.

Concluindo, o melhor algoritmo para esta rede é o *Greedy*. O *Greedy* gera soluções muito melhores que o *Random* e o *Multi-Start* não é capaz de as melhorar. O *Greedy*, por a rede considerada no exercício ser de pequenas dimensões(em número de nós(10) e de ligações(16)), obtém solução similares ao *Multi-Start* (pelo menos no tempo tempo de pesquisa considerado, 10 segundos) e obtém mais soluções(maior em 1 ordem de grandeza).

Capítulo 3

Tarefa 2

3.1 alínea A

Código

O objectivo desta alínea era criar e correr um algoritmo que computasse uma solução de *routing* simétrica de um único caminho para cada fluxo da rede de modo a suportar um serviço que minimiza-se o consumo de energia da rede.

Por isso foi desenvolvido um algoritmo que durante 10 segundos escolhe aleatoriamente caminhos de *routing* para cada fluxo de modo a obter aquele que minimiza o consumo de energia da rede. O algoritmo foi corrido para três situações diferentes: considerando todos os caminhos de *routing*, ou apenas os 10 menores caminhos ou apenas os 5 menores caminhos.

O que difere neste caso da primeira tarefa é que o objectivo agora é ter em conta a energia do *link* enquanto que no primeiro caso era a *load*. Todo o resto do processo é exactamente igual. Por isso no código apresentado a seguir apenas é apresentado o cálculo que difere da primeira alínea.

```
%% 2.a) % Random Strategy
( ... )
% Iterate through the serveral limits
for limit = limits
    t= tic;
    bestEnergy= inf; % Variable to store best energy
    sol= zeros(1,nFlows); % Vector to store solution
    allValues= [];

    %During 10 seconds run random algorithm
    while toc(t)<10
        ( ... )
        %Calculate load of the solution
        Loads= calculateLinkLoads(nNodes,Links,T,sP,sol);
        load= max(max(Loads(:,3:4)));
        %Link Load must be smaller than 10 Gbps
```

```
            if load≤10
                energy = 0;
                for a=1:nLinks
                    %If link is being used, it's load must be > 0
                    if Loads(a,3)+Loads(a,4)>0
                        % energy consumption is proporcinal to the link length
                        energy = energy + L(Loads(a,1),Loads(a,2));
                    end
                end
            else
                energy=inf;
            end
            allValues= [allValues energy];
            %Update best energy if necessary
            if energy<bestEnergy
                bestSol= sol;
                bestEnergy= energy;
            end
        end
        %Plot values and print results
        plot(sort(allValues));
        hold on;
        fprintf('%d best paths\n', limit);
        fprintf('   Best energy = %.1f\n', bestEnergy);
        fprintf('   No. of solutions = %d\n', length(allValues));
        aux = allValues(allValues ≠ inf);
        fprintf('   Av. quality of solutions = %.1f\n' ,mean(aux));
    end
    ( ... )
```

Resultados

Analisando os resultados obtidos reparamos que a qualidade das soluções vai melhorando à medida que reduzimos o número de caminhos de entre os quais escolher uma solução. O mesmo acontece com a energia calculada a partir do caminho solução. Em relação ao número de soluções é observável que são bastante parecidos em ambos os casos. Uma particularidade interessante observável é o facto de o número de soluções que aparecem no gráfico não corresponder ao que é apresentado na figura Figura 3.3. Analisando o código percebemos que se na solução encontrada a *link load* for maior que 10 Gbps então a mesma é inválida, de forma tal que o valor armazenado na variável *energy* é *inf*. A apresentação destes valores nos gráficos é ignorada pelo facto de ser impossível representar num gráfico o valor infinito. Deste modo percebe-se assim que considerando todos os caminhos vamos obter mais soluções inválidas pelo facto de estarmos a aumentar o espaço de escolha. Ao aumentar o número de soluções inválidas significa que menos soluções teremos no gráfico pelo facto da grande maioria delas ter o valor *inf*.

Menos caminhos de *routing* de entre os quais escolher oferecem melhores resultados pelo facto da escolha ser aleatória e reduzirmos a possibilidade de encontrarmos soluções inválidas. Isto justifica também a situação oposta, quanto mais se aumenta o espaço de escolha, maior é a probabilidade de ser seleccionada uma solução inválida (porque a escolha é aleatória). É im-

portante perceber também que a energia de uma ligação é proporcional ao seu comprimento. Logo quando são calculados os *shortest paths*, estes encontram-se nas primeiras posições. Assim ao escolher dos 5 ou 10 menores caminhos garantimos que estamos a escolher as melhores opções que irão dar menor energia. E quanto mais aumentamos o número de possibilidades, mais soluções com pior qualidade estamos a adicionar ao espaço de resultados passiveis o que decresce não só a qualidade média das soluções mas que pode impedir também a possibilidade de uma escolha acertada.

Em termos do número de soluções obtidas é claro que pelo facto do algoritmo escolher de forma aleatória este seja bastante parecido nos 3 casos pois não existe grande diferença no processamento de uns para outros. Se corrermos várias vezes o algoritmo estes valores podem oscilar.

```
RANDOM:
Inf best paths
   Best energy = 2232.0
   No. of solutions = 138054
   Av. quality of solutions = 3057.7
10 best paths
   Best energy = 1553.0
   No. of solutions = 139910
   Av. quality of solutions = 2846.7
5 best paths
   Best energy = 1360.0
   No. of solutions = 138782
   Av. quality of solutions = 2646.4
```

Figura 3.1: Resultado da alínea 2.a).

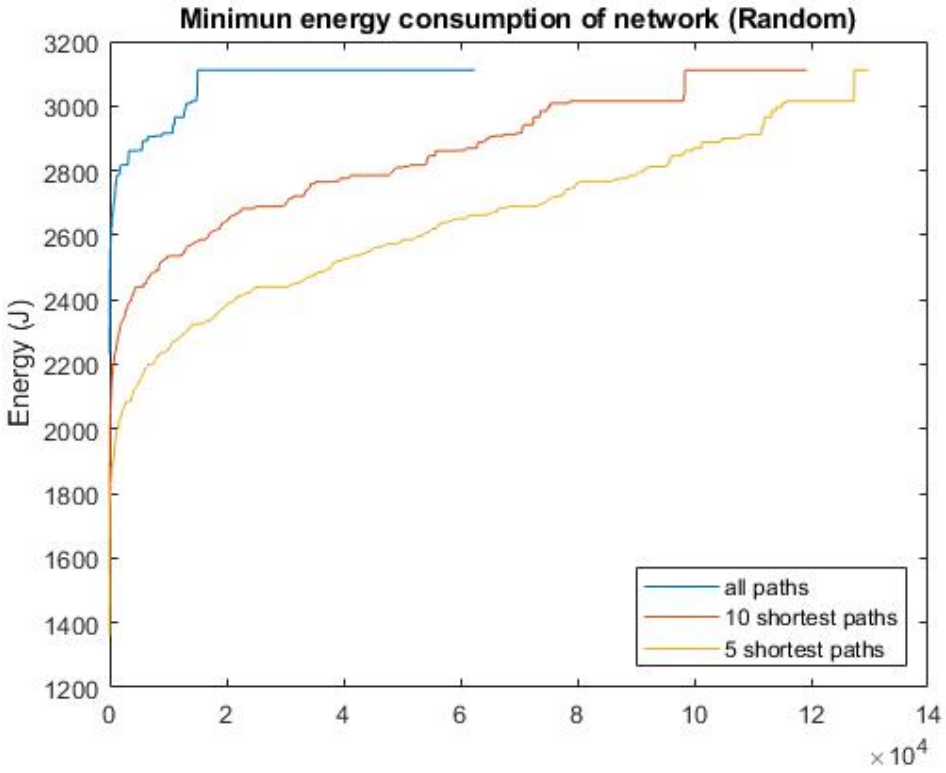

Figura 3.2: Gráfico resultante da execução da alínea 2.a).

3.2 alínea B

Código

Nesta alínea era pedido o mesmo que na anterior mas desta vez utilizando o algoritmo *Greedy Randomized*. Este algoritmo funciona da mesma maneira que aquele que foi descrito na alínea 1.c no entanto desta vez está pronto para em vez de escolher soluções com menor *link load*, escolher soluções com minimização do consumo de energia. Novamente apenas é apresentadas as alterações ao código apresentado na solução 1.c.

```
%% 2.b) % Greedy Randomized Strategy
( ... )
for limit = limits
    t= tic;
    bestEnergy= inf;
    sol= zeros(1,nFlows);
    allValues= [];
    % Run algorithm during 10 seconds
    while toc(t)<10
        ( ... )
        for i= ax2
```

```
            ( ... )
            for k = 1:n
                sol(i)= k;
                Loads= calculateLinkLoads(nNodes,Links,T,sP,sol);
                load= max(max(Loads(:,3:4)));
                if load≤10
                    energy = 0;
                    for a=1:nLinks
                        if Loads(a,3)+Loads(a,4)>0
                            %Energy = Energy + Link Length
                            energy = energy + L(Loads(a,1),Loads(a,2));
                        end
                    end
                else
                    energy=inf;
                end
                % Update best solution variables if necessary
                if energy < best
                    k_best = k;
                    best = energy;
                end
            end
            % Update on solution vector best choice
            sol(i) = k_best;
        end
        Loads= calculateLinkLoads(nNodes,Links,T,sP,sol);
        load= max(max(Loads(:,3:4)));
        if load≤10
            energy = 0;
            for a=1:nLinks
                if Loads(a,3)+Loads(a,4)>0
                    energy = energy + L(Loads(a,1),Loads(a,2));
                end
            end
        else
            energy=inf;
        end
        % Add to all Values and update best energy if necessary
        allValues= [allValues energy];
        if energy <bestEnergy
            bestSol= sol;
            bestEnergy= energy;
        end
    end
    ( ... )
end
( ... )
```

Resultados

Analisando os resultados obtidos concluímos que à medida que se reduz o número de caminhos de entre os quais se escolhe um solução, se aumenta o número de soluções. Por sua vez a qualidade das soluções decresce com a redução do número de caminhos, assim como a

qualidade da própria solução obtida.

Tal como no exercício anterior, este algoritmo *Greedy Randomized* escolhe no geral a solução mais adequada no momento. Isto significa que aumentando o espaço de soluções não estamos a introduzir soluções inválidas na pesquisa, mas sim uma maior precisão na solução obtida pelo facto de ser ter em conta mais valores, e dos mesmos escolher a melhor solução possível. E como o algoritmo *Greedy* escolhe a melhor solução no momento significa então que soluções inválidas (em que a *link load* é maior que 10 Gbps) são descartadas, logo a inconsistência entre o gráfico obtido e os valores estatísticos obtidos na alínea anterior não se verificam neste caso.

Em relação ao número de soluções já percebemos na alínea anterior que quantos mais caminhos existem por onde escolher, significa que mais processamento o algoritmo de *Greedy Randomized* irá ter de fazer, o que consequentemente deixa menos tempo para a procura de novas soluções. Logo percebe-se assim o porquê de conseguirmos ter muitas soluções com apenas 5 caminhos em relação à escolha de entre todos os caminhos (a primeira requer muito menos tempo de procura de solução que a outra).

```
GREEDY RANDOMIZED:
Inf best paths
   Best energy = 1235.0
   No. of solutions = 4367
   Av. quality of solutions = 1393.6
10 best paths
   Best energy = 1235.0
   No. of solutions = 14919
   Av. quality of solutions = 1416.9
5 best paths
   Best energy = 1303.0
   No. of solutions = 31625
   Av. quality of solutions = 1512.7
```

Figura 3.3: Resultado da alínea 2.b).

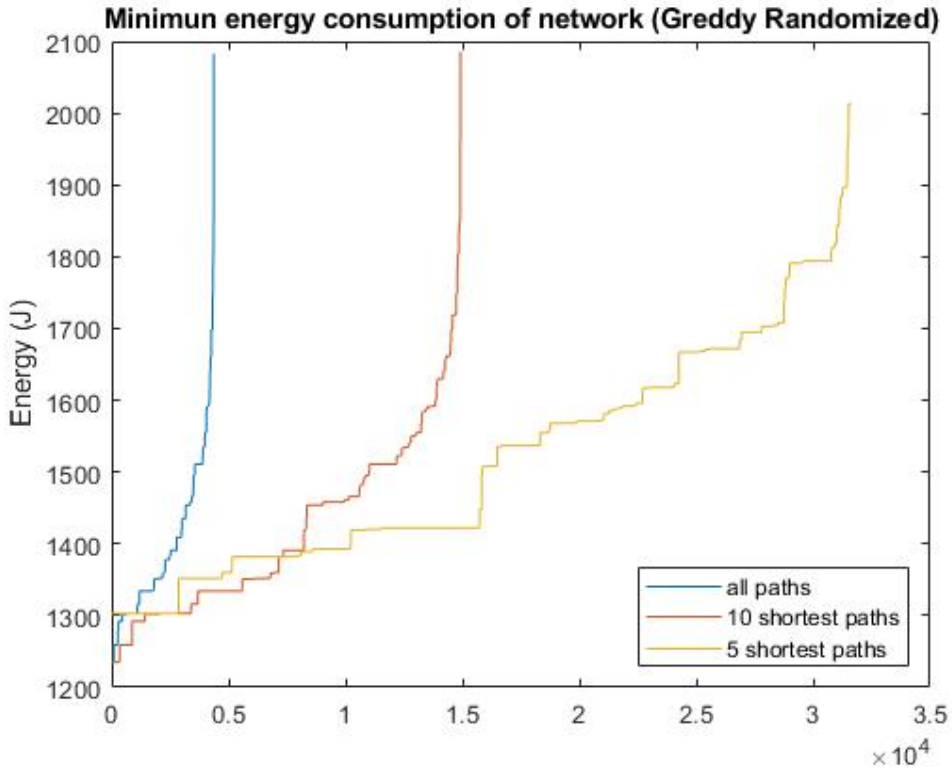

Figura 3.4: Gráfico resultante da execução da alínea 2.b).

3.3 alínea C

Neste caso novamente o que se pretendia era o mesmo que na alínea 1.d) mas desta vez considerando a minimização do consumo de energia da rede e não a *worst link load*. Para isso pretendia-se que se utiliza-se o algoritmo *Multi Start Hill Climbing*. Novamente foram utilizadas duas funções auxiliares para facilitar a legibilidade do código deste algoritmo. São novamente apresentadas apenas as alterações que foram feitas ao código da alínea 1.d.

Código

```
( ...)
for limit = limits
    globalBestEnergy= inf;
    ( ... )
    while toc(t) < 10
        %Greedy Randomized Solution
        [bestSol,bestEnergy] = greedyRandomizedEnergy(nFlows,nSP, nNodes, ...
            Links, T, L, sP, limit);
        repeat = true;
        while repeat
```

```
            %Iterate through all values of the solution (to calculate best ...
                neighbor)
            neighborBest= inf;
            for i=1:nFlows
                [nS, nE]= BuildNeighborEnergy(bestSol,i, sP, nSP, Links, ...
                    nNodes, T, L, bestEnergy);
                if nE < neighborBest
                    neighborBest= nE;
                    neighborSol= nS;
                end
            end
            if neighborBest < bestEnergy
                bestSol= neighborSol;
                bestEnergy= neighborBest;
            else
                repeat= false;
            end
        end
        allValues= [allValues bestEnergy];
        if bestEnergy < globalBestEnergy
            globalBestEnergy= bestEnergy;
            globalSol= bestSol;
        end
    end
    ( ... )
end
( ... )
```

Função *BestNeighborEnergy* que é uma função auxiliar para uma dada solução e um certo index dessa solução calcula as soluções vizinhas a essa mesma solução e devolve a melhor que for encontrada. Caso não seja encontrada nenhuma melhor é devolvida a actual. Segue a mesma lógica que a que foi descrita na alínea 1.d) mas adpatada ao consumo de energia da rede.

```
function [nSol, nEn] = BuildNeighborEnergy(bestSol,i, sP, nSP, Links, ...
    nNodes, T, L, bestEnergy)
( ... )
%   bestEnergy - best energy found with  greedy randomized solution

nFlows= size(bestSol);
nSol= bestSol;
nEn= bestEnergy;
nLinks= size(Links,1);

k=0;
value=0;

for n=1:nFlows
    if n≠i
        newNeighbor= bestSol;
        for j=1:nSP(n)
            if j≠bestSol(n)
                newNeighbor(n)= j;
                Loads= calculateLinkLoads(nNodes,Links,T,sP,newNeighbor);
```

```
                load= max(max(Loads(:,3:4)));
                if load ≤ 10
                    energy= 0;
                    for a= 1:nLinks
                        if Loads(a,3)+Loads(a,4)>0
                            energy= energy + L(Loads(a,1),Loads(a,2));
                        end
                    end
                else
                    energy= inf;
                end
                if energy < nEn
                    k= n;
                    value= j;
                end
            end
        end
    end
end
if k>0
    nSol(k)= value;
end


end
```

Função *GreedyRandomizedEnergy* que implementa o algoritmo da alínea 2.b mas que foi criada para tornar o código do algoritmo *Multi Start Hill Climbing* mais legível.

```
function [sol,eN] = greedyRandomizedEnergy(nFlows,nSP, nNodes, Links, T, ...
    L, sP, limit)
( ... )
for i= ax2
    ( ... )
    for k= 1:n
        sol(i)= k;
        Loads= calculateLinkLoads(nNodes,Links,T,sP,sol);
        load= max(max(Loads(:,3:4)));
        if load≤10
            energy = 0;
            for a=1:nLinks
                if Loads(a,3)+Loads(a,4)>0
                    energy = energy + L(Loads(a,1),Loads(a,2)); %a energia ...
                        e igual a energia + o comprimento do link (L -> ...
                        comprimento do link)
                end
            end
        else
            energy=inf;
        end
        if energy < best
            k_best= k;
            best= energy;
        end
```

```
        end
        if k_best>0
            sol(i) = k_best;
        else
            break;
        end
    end
    eN= best;

    end
```

Resultados

Analisando os resultados obtidos percebemos que o número de soluções aumenta à medida que o número de caminhos de entre os quais escolher a solução reduz. Ao mesmo tempo também se percebe que a qualidade da solução encontrada e qualidade média das soluções no geral também piora com a redução do número de caminhos de *routing*.

Novamente, contrariamente à primeira alínea, os valores apresentados no gráfico correspondem aos que forem recolhidos na Figura 3.5. Isto acontece pois como já vimos o algoritmo *Greedy Randomized* recolhe apenas soluções válidas, e após essa recolha é feita uma pesquisa pelos vizinhos dessa solução, sendo que apenas são consideradas soluções válidas (cuja *link load* é maior que 10 Gbps). Logo não são consideradas nenhumas soluções cuja energia tenha o valor de *inf* pois apenas são escolhidas soluções válidas que sejam melhores que a atual.

Em termos da qualidade das soluções já verificámos no exercício anterior que dependem da qualidade devolvida pelo *Greedy Randomized* pelo facto de ser feita uma pesquisa pelas soluções vizinhas. E percebemos também na alínea anterior que o *Greedy* irá devolver melhor qualidade quando pode pesquisar por todos os caminhos e pior qualidade quando pesquisa por um número mais reduzido de caminhos. Quanto melhor a qualidade melhor a possibilidade do algoritmo de *Multi Start* encontrar vizinhos também com ligeira melhor qualidade.

```
MULTI START HILL CLIMBING:
Inf best paths
   Best energy = 1235.0
   No. of solutions = 2417
   Av. quality of solutions = 1396.4
10 best paths
   Best energy = 1235.0
   No. of solutions = 3422
   Av. quality of solutions = 1410.0
5 best paths
   Best energy = 1235.0
   No. of solutions = 3261
   Av. quality of solutions = 1470.2
```

Figura 3.5: Resultado da alínea 2.c).

Em relação ao número de soluções obtidas já percebemos que o *Multi Start* terá mais tempo para pesquisar soluções quando tem menos processamento para fazer por cada solução. O *Multi Start* por sua vez tem mais tempo para procurar soluções quando o *Greedy* também o tem e isto acontece quando o número de caminhos a escolher é menor. Analisando e comparando com a alínea anterior, podemos ver que o *Multi Start* só melhorou o valor obtido pelo *Greedy* no caso em que escolheu dos 5 menores caminhos. Em todos os outros, apesar da qualidade das soluções ser melhor, não se verificaram melhorias.

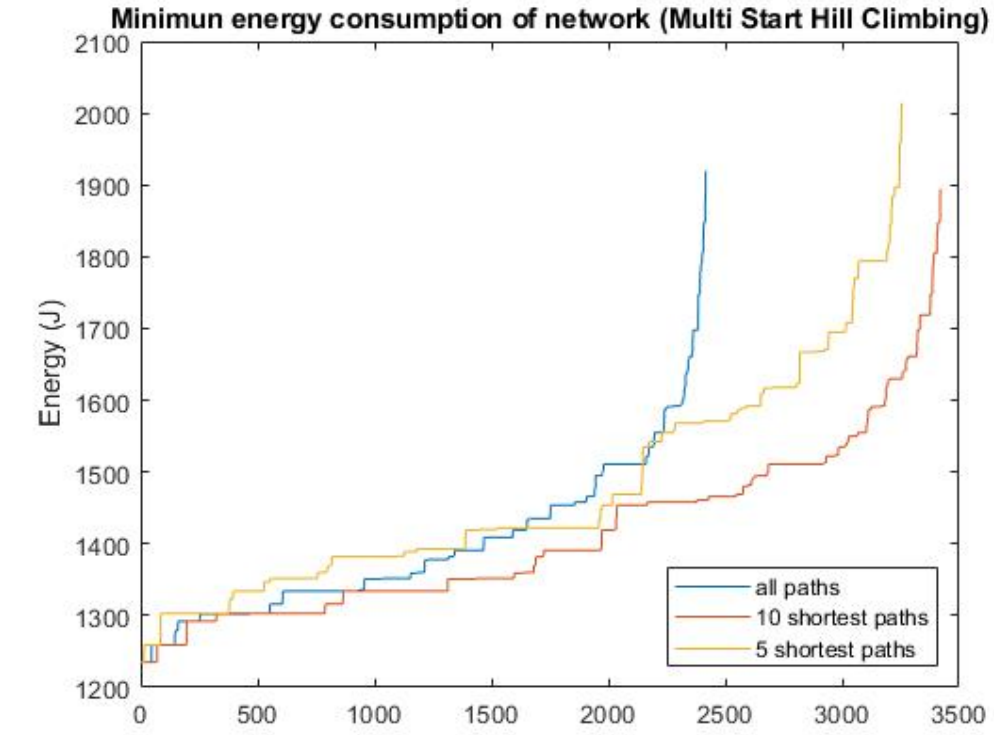

Figura 3.6: Gráfico resultante da execução da alínea 2.c).

3.4 alínea D

Comparando os três algoritmos iremos tirar conclusões bastante parecidas com a alínea 1.e). O algoritmo *Random* é computacionalmente pouco intensivo, no entanto a qualidade de soluções que oferece não é tão fiável. Os outros dois algoritmos já são computacionalmente muito mais intensivos, oferecendo poucas soluções, mas de qualidade (pelo facto de perderem muito tempo a procurarem resultados ideais). O algoritmo *Multi Start* é o que oferece um número menor de soluções, no entanto essas poucas soluções têm uma qualidade bastante elevada quando comparadas com as do *Random*.

Concluindo com base tanto no resultado do algoritmo *Multi Start* para o caso da *link load* como para o da energia, verificamos que em ambos os casos este algoritmo não foi capaz de melhorar a solução oferecida pelo *Greedy*. Acrescentando o facto de ser computacionalmente mais intensivo, e não ser capaz de melhorar a solução *Greedy*, concluimos que este algoritmo não é tão bom como o *Greedy*, que por sua vez visita um número muito maior de soluções, não é tão computacionalmente exigente e apresenta resultados bastante semelhantes aos do *Multi start*.

Capítulo 4

Tarefa 3

4.1 alínea A

Código

Nesta alínea o objectivo pretendido era calcular, para cada fluxo um dos seus *routing paths* dado pelo caminho com maior disponibilidade. Para tal efeito foi usada a função fornecida pelo professor *calculatePaths* que por sua vez irá devolver os caminhos como maior disponibilidade. Para isto funcionar de forma correta teve de ser dado como argumento de entrada a matriz *logA*. Esta matriz é dada pelo logaritmo dos elementos da matriz A e representa uma forma uniforme de inverter os pesos das disponibilidades de cada link. Deste modo as soluções que vão ser escolhidas são aquelas que têm menor peso, logo maior disponibilidade.

```
function [sP nSP]= calculatePaths(L,T,n)
    nFlows= size(T,1);
    nSP= zeros(1,nFlows);
    for i=1:nFlows
        [shortestPath, totalCost] = kShortestPath(L,T(i,1),T(i,2),n);
        sP{i}= shortestPath;
        nSP(i)= length(totalCost);
    end
end
```

```
%% 3.a)
% Compute up to 1 paths for each flow:
n= 1;
fprintf("Alinea A\n");
[sP, nSP]= calculatePaths(logA,T,n);

for n = 1 : nFlows
    path = sP{n}{1};
    fprintf('flow %d, %2d <-> %2d:', n, T(n,1), T(n,2))
    disp(path)
end
```

Resultados

```
Alínea A
flow 1,  1 <->  3:     1     2     3
flow 2,  1 <->  4:     1     5     4
flow 3,  2 <->  7:     2     4     5     7
flow 4,  3 <->  4:     3     4
flow 5,  4 <->  9:     4     8     9
flow 6,  5 <->  6:     5     7     8     6
flow 7,  5 <->  8:     5     7     8
flow 8,  5 <->  9:     5     7     9
flow 9,  6 <-> 10:     6    10
```

Figura 4.1: Resultado da alínea 3.a).

4.2 alínea B

Para esta alínea o objectivo era para cada fluxo, calcular outro caminho de *routing* dado pelo caminho com maior disponibilidade que fosse disjunto do primeiro calculado. De notar que a função best2Paths é similar à função kShortestPaths disponibilizada.

Código

```
%% 3.b)
clc
fprintf("Alinea B\n");
[sP, nSP] = best2Paths(logA, T);
avail = zeros(1, nFlows);
avail2nd = zeros(1, nFlows);
...
% prints
...
```

```
function [sP nSP]= best2Paths(L,T)
    nFlows= size(T,1);
    nSP= zeros(1,(2*nFlows));
    for i=1:nFlows
        [shortestPath, totalCost] = kShortestPath(L,T(i,1),T(i,2),1);
        path = shortestPath{1};
        sP{i}= shortestPath;
        nSP(i)= length(totalCost);
        auxL = L;
        for j = 1: (length(path) - 1)
            auxL(path(j), path(j+1)) = inf;
            auxL(path(j+1), path(j)) = inf;
        end
        [shortestPath, totalCost] = kShortestPath(auxL,T(i,1),T(i,2),1);
        sP{nFlows + i} = shortestPath;
        nSP(nFlows + i) = length(totalCost);
    end
end
```

Resultados

```
Alínea B
flow 1 | 1  <->  3:
         best: 1  2  3              | availability: 0.9965
best disjoint: 1  5  4  3      | availability: 0.9964
flow 2 | 1  <->  4:
         best: 1  5  4              | availability: 0.9979
best disjoint: 1  2  4      | availability: 0.9974
flow 3 | 2  <->  7:
         best: 2  4  5  7              | availability: 0.9976
best disjoint: 2  3  8  7      | availability: 0.9955
flow 4 | 3  <->  4:
         best: 3  4              | availability: 0.9985
best disjoint: 3  2  4      | availability: 0.9979
flow 5 | 4  <->  9:
         best: 4  8  9              | availability: 0.9976
best disjoint: 4  5  7  9      | availability: 0.9972
flow 6 | 5  <->  6:
         best: 5  7  8  6              | availability: 0.9969
best disjoint: 5  4  3  6      | availability: 0.9965
flow 7 | 5  <->  8:
         best: 5  7  8              | availability: 0.9977
best disjoint: 5  4  8      | availability: 0.9974
flow 8 | 5  <->  9:
         best: 5  7  9              | availability: 0.9985
best disjoint: 5  4  8  9      | availability: 0.9962
flow 9 | 6  <-> 10:
         best: 6  10              | availability: 0.9994
best disjoint: 6  8  9  10      | availability: 0.9959
average availability for best paths: 0.9978
average availability for 2nd best paths disjoint with best path: 0.9967
```

Figura 4.2: Resultado da alínea 3.b).

4.3 alínea C

Código

Nesta alínea o objectivo era calcular a largura de banda necessária em cada direcção de cada *link* para suportar todos os fluxos com protecção 1+1 usando os pares de *links* com caminhos disjuntos computados na anteriormente.

```
%% 3.c)
fprintf("Alinea C");
[sP, nSP] = best2Paths(logA, T);
p1st = cell(nFlows, 1);
```

```
p2nd = cell(nFlows, 1);
for i = 1:nFlows
    p1st{i} = sP{i};
    p2nd{i} = sP{i + nFlows};
end
sol= ones(1, nFlows);
% best
Loads1st = calculateLinkLoads(nNodes,Links,T,p1st,sol)
% 2nd best disjoint
Loads2nd = calculateLinkLoads(nNodes,Links,T,p2nd,sol);

Loads = [Loads1st(:,1), Loads1st(:,2), (Loads1st(:,3) + Loads2nd(:,3)), ...
    (Loads1st(:,4) + Loads2nd(:,4))];
...
% prints
...
```

Resultados

```
Alínea C
Proteção 1+1:
link nº1  || 1  ->  2 : 1.70  Gb          | 2  ->  1 : 1.50  Gb
link nº2  || 1  ->  5 : 1.70  Gb          | 5  ->  1 : 1.50  Gb
link nº3  || 2  ->  3 : 5.50  Gb          | 3  ->  2 : 4.90  Gb
link nº4  || 2  ->  4 : 5.50  Gb          | 4  ->  2 : 4.10  Gb
link nº5  || 3  ->  4 : 4.90  Gb          | 4  ->  3 : 4.30  Gb
link nº6  || 3  ->  6 : 1.20  Gb          | 6  ->  3 : 1.50  Gb
link nº7  || 3  ->  8 : 2.40  Gb          | 8  ->  3 : 1.50  Gb
link nº8  || 4  ->  5 : 10.80 Gb > 10 Gb  | 5  ->  4 : 10.30 Gb > 10 Gb
link nº9  || 4  ->  8 : 4.70  Gb          | 8  ->  4 : 6.60  Gb
link nº10 || 5  ->  7 : 8.30  Gb          | 7  ->  5 : 9.60  Gb
link nº11 || 6  ->  8 : 2.90  Gb          | 8  ->  6 : 2.80  Gb
link nº12 || 6  -> 10 : 1.40  Gb          | 10 ->  6 : 1.60  Gb
link nº13 || 7  ->  8 : 4.80  Gb          | 8  ->  7 : 6.40  Gb
link nº14 || 7  ->  9 : 2.60  Gb          | 9  ->  7 : 4.10  Gb
link nº15 || 8  ->  9 : 4.00  Gb          | 9  ->  8 : 5.70  Gb
link nº16 || 9  -> 10 : 1.40  Gb          | 10 ->  9 : 1.60  Gb
Sum of all links in both directions: 131.80 Gb
```

Figura 4.3: Resultado da alínea 3.c).

4.4 alínea D

Neste caso pretendia-se que fosse calculada a largura de banda para cada *link* para suportar todos os fluxos com protecção 1:1 usando os pares de *links* com caminhos disjuntos computados na anteriormente.

Código

```
% 3.d)
fprintf("Alinea D");
Loads = calculateLinkLoads1to1(nNodes, Links, T, p1st, p2nd);
..
% prints
...
```

Resultados

```
Alínea D
Proteção 1:1:
link nº1  || 1  ->  2 : 1.70  Gb        | 2  ->  1 : 1.50  Gb
link nº2  || 1  ->  5 : 1.70  Gb        | 5  ->  1 : 1.50  Gb
link nº3  || 2  ->  3 : 3.40  Gb        | 3  ->  2 : 3.40  Gb
link nº4  || 2  ->  4 : 4.80  Gb        | 4  ->  2 : 3.60  Gb
link nº5  || 3  ->  4 : 3.90  Gb        | 4  ->  3 : 3.30  Gb
link nº6  || 3  ->  6 : 1.20  Gb        | 6  ->  3 : 1.50  Gb
link nº7  || 3  ->  8 : 2.40  Gb        | 8  ->  3 : 1.50  Gb
link nº8  || 4  ->  5 : 6.90  Gb        | 5  ->  4 : 5.60  Gb
link nº9  || 4  ->  8 : 4.70  Gb        | 8  ->  4 : 6.60  Gb
link nº10 || 5  ->  7 : 8.30  Gb        | 7  ->  5 : 9.60  Gb
link nº11 || 6  ->  8 : 2.90  Gb        | 8  ->  6 : 2.80  Gb
link nº12 || 6  -> 10 : 1.40  Gb        | 10 ->  6 : 1.60  Gb
link nº13 || 7  ->  8 : 4.80  Gb        | 8  ->  7 : 6.40  Gb
link nº14 || 7  ->  9 : 2.60  Gb        | 9  ->  7 : 4.10  Gb
link nº15 || 8  ->  9 : 2.60  Gb        | 9  ->  8 : 4.10  Gb
link nº16 || 9  -> 10 : 1.40  Gb        | 10 ->  9 : 1.60  Gb
Sum of all links in both directions: 113.40 Gb
```

Figura 4.4: Resultado da alínea 3.d).

4.5 alínea E

O mecanismo de protecção 1+1 reitera que todos os links sejam capazes de suportar todos os fluxos de dados que circulem entre eles, quer fluxos primários quer fluxos de protecção. Ora aplicando este mecanismo com os melhores pares disjuntos de todos os fluxos (calculados de forma independente) obtivemos os resultados da Figura 4.3. Podemos observar que, este mecanismo exige muita largura de banda na rede. Podemos ainda observar que para o link número 8 em ambas as direções não suporta este mecanismo pois a largura de banda calculada excede o limite físico da rede. Assim este mecanismo não podia ser implementado nesta rede por não garantir total protecção para a mesma (pelo menos com esta configuração de caminhos escolhido para os fluxos).

Destacar ainda que as 3 ligações com maior carga em qualquer uma das direcções são a 8, 9 e a 10. Isto justifica-se por a ligação 8, e 9 serem no centro da rede(considerando uma ligação entre routers nos extremos da rede uma ligação na periferia na rede). A ligação 10 apesar de ser na periferia da rede é a mais curta em comprimento e consequentemente a com maior *availability* e por conseguinte mais vezes escolhida por apresentar melhor *availability* que seus *peers*. O link número 10 ainda pertence ao caminho de menor distância entre os nós do fluxo número 3, nó 2 e nó 7. Este fluxo é dos que mais largura de banda necessita, sendo que na direcção nó 2 para o nó 7, necessita de 2.4 Gb.

Para o mecanismo de protecção 1:1 é preciso garantir que que cada link tem largura de banda tal que suporte o fluxo com a maior largura de banda que transite nessa ligação. Este mecanismo pode garantir total protecção da rede. Para a figura Figura 4.4 observamos que neste caso não foi identificado nenhum link que excedesse o limite de 10 Gb em qualquer das direções. De notar que pelas razões já explicadas os links 8, 9 e 10 voltam a ser os que necessitam de maior largura de banda em qualquer um dos sentidos.

Concluindo, era de esperar que a proteção 1:1 necessitasse de menor ou igual largura de banda em todas as ligações para a mesma configuração. No entanto em caso de falha não garante um tempo de recuperação tão bom quanto o mecanismo 1+1.

Capítulo 5

Tarefa 4

5.1 alínea A

Neste exercício seguindo a indicação do enunciado, calculamos os k *paths* com maior disponibilidade com a função *kShortestPath.m* para cada flow da rede. E posteriormente calculamos os k *paths* disjunto do k anterior para cada fluxo. Esta lógica foi implementada na função *bestKpaths.m*.

A variável *sP1* tem os k caminhos com maior disponibilidade para cada fluxo e a variável *sP2* os respectivos pares com melhor *availability* disjunto do primeiro.

```
function [sP1 nSP1 sP2 nSP2]= bestKpaths(L,T,k)
    nFlows= size(T,1);
    nSP1= zeros(1,nFlows);
    % e k = 10 most available routing paths provided by the network to ...
       each traffic flow
    for f = 1:nFlows
        [shortestPath, totalCost] = kShortestPath(L,T(f,1),T(f,2),k);
        sP1{f}= shortestPath;
        nSP1(f)= length(totalCost);
    end
    nSP2 = zeros(1,nFlows);
    for f = 1:nFlows
        flowK_shortest = sP1{f};
        for x = 1:nSP1(f) % k
            path = flowK_shortest{x};
            auxL = L;
            for j = 1: (length(path) - 1)
                auxL(path(j), path(j+1)) = inf;
                auxL(path(j+1), path(j)) = inf;
            end
            [shortestPath, totalCost] = kShortestPath(auxL,T(f,1),T(f,2),1);
            if ¬isempty(shortestPath)
                aux{x} = shortestPath{1};
            else
                aux{x} = [];
            end
```

```
            nSP2(f) = length(totalCost);
        end
        sP2{f} = aux;
    end
end
```

5.2 alínea B

Nesta alínea foi desenvolvido um algoritmo *Multi Start Hill Climbing* que usa os 10 pares de *links* disjuntos calculados na alínea anterior.

```
%% 4.a)
k= 10;
[sP1, nSP1, sP2, nSP2] = bestKpaths(logA, T, k);

% 4.b)
allValues = [];

globalbestcell1 = 0;
globalbestcell2 = 0;

globalbestpairindex = [];
globalbestload = inf;
t=tic;
while toc(t) < 30
    ax2 = randperm(nFlows);
    bestcell1 = cell(nFlows, 1);
    bestcell2 = cell(nFlows, 1);
    for xx= 1:nFlows
        bestcell1{xx} = {[]};
        bestcell2{xx} = {[]};
    end
    bestpairindex = [];
    for i= ax2
        k_best = 0;
        best = inf;
        sp1K = sP1{i};
        sp2K = sP2{i};
        for j= 1:k
            path1 = {sp1K{j}};
            path2 = {sp2K{j}};
            cell1 = bestcell1;
            cell2 = bestcell2;
            cell1{i} = path1;
            cell2{i} = path2;
            Loads= calculateLinkLoads1to1(nNodes,Links,T,cell1, cell2);
            load= max(max(Loads(:,3:4)));
            if load < best
                k_best = j;
                best = load;
            end
        end
```

```
            bestcell1{i} = {sp1K{k_best}};
            bestcell2{i} = {sp2K{k_best}};
            bestpairindex = [bestpairindex k_best];
        end
        repeat = true;

        neighborBestLoad = inf;
        neighborBestcell1 = 0;
        neighborBestcell2 = 0;
        while repeat
            for i= 1:nFlows
                kk = 0;
                value = 0;
                for n=1:nFlows
                    if n≠ i
                        newNeighbor1= bestcell1;
                        newNeighbor2= bestcell2;
                        for j= 1:k
                            if j≠ bestpairindex(n)
                                newNeighbor1{n} = sP1{n}(j);
                                newNeighbor2{n} = sP2{n}(j);
                                Loads= calculateLinkLoads1to1(nNodes, Links, ...
                                    T, newNeighbor1, newNeighbor2);
                                load= max(max(Loads(:,3:4)));
                                if load < neighborBestLoad
                                    bestcell1 = newNeighbor1;
                                    bestcell2 = newNeighbor2;
                                    neighborBestLoad = load;
                                    bestpairindex(n) = j;
                                end
                            end
                        end
                    end
                end
            end
            if neighborBestLoad < best
                best = neighborBestLoad;
                neighborBestcell1 = bestcell1;
                neighborBestcell2 = bestcell2;
            else
                repeat = false;
            end
        end
        allValues = [allValues best];
        if best < globalbestload
            globalbestload = best;
            globalbestcell1 = neighborBestcell1;
            globalbestcell2 = neighborBestcell2;
            globalbestpairindex = bestpairindex;
        end
    end
    ...
    % prints
    ...
```

Resultados

```
Best solution found
:flow 1,  1 <->  3:
          best path: 1  2  3              | availability: 0.9965
    protection path: 1  5  4  3      | availability: 0.9964
flow 2,  1 <->  4:
          best path: 1  2  4              | availability: 0.9974
    protection path: 1  5  4      | availability: 0.9979
flow 3,  2 <->  7:
          best path: 2  4  8  7              | availability: 0.9963
    protection path: 2  1  5  7      | availability: 0.9968
flow 4,  3 <->  4:
          best path: 3  4              | availability: 0.9985
    protection path: 3  2  4      | availability: 0.9979
flow 5,  4 <->  9:
          best path: 4  8  9              | availability: 0.9976
    protection path: 4  5  7  9      | availability: 0.9972
flow 6,  5 <->  6:
          best path: 5  1  2  3  6              | availability: 0.9950
    protection path: 5  7  8  6      | availability: 0.9969
flow 7,  5 <->  8:
          best path: 5  4  3  8              | availability: 0.9959
    protection path: 5  7  8      | availability: 0.9977
flow 8,  5 <->  9:
          best path: 5  7  9              | availability: 0.9985
    protection path: 5  4  8  9      | availability: 0.9962
flow 9,  6 <-> 10:
          best path: 6  10              | availability: 0.9994
    protection path: 6  8  9  10      | availability: 0.9959
average availability for best paths: 0.9972
average availability for 2nd best paths disjoint with best path: 0.9970

Best load = 5.90 Gbps
No. of solutions = 172
Averg. quality of solutions = 6.58 Gbps
Sum of all links in both directions: 115.70 Gb

Proteção 1:1 para a Best solution:
link n°1  || 1  ->  2 : 4.40  Gb         | 2  ->  1 : 5.40  Gb
link n°2  || 1  ->  5 : 4.60  Gb         | 5  ->  1 : 3.20  Gb
link n°3  || 2  ->  3 : 4.30  Gb         | 3  ->  2 : 4.90  Gb
link n°4  || 2  ->  4 : 5.50  Gb         | 4  ->  2 : 4.10  Gb
link n°5  || 3  ->  4 : 5.90  Gb         | 4  ->  3 : 5.20  Gb
link n°6  || 3  ->  6 : 1.20  Gb         | 6  ->  3 : 1.50  Gb
link n°7  || 3  ->  8 : 2.10  Gb         | 8  ->  3 : 2.50  Gb
link n°8  || 4  ->  5 : 4.40  Gb         | 5  ->  4 : 4.30  Gb
link n°9  || 4  ->  8 : 5.00  Gb         | 8  ->  4 : 5.60  Gb
link n°10 || 5  ->  7 : 5.00  Gb         | 7  ->  5 : 5.60  Gb
link n°11 || 6  ->  8 : 1.50  Gb         | 8  ->  6 : 1.60  Gb
link n°12 || 6  -> 10 : 1.40  Gb         | 10 ->  6 : 1.60  Gb
link n°13 || 7  ->  8 : 3.60  Gb         | 8  ->  7 : 4.90  Gb
link n°14 || 7  ->  9 : 2.60  Gb         | 9  ->  7 : 4.10  Gb
link n°15 || 8  ->  9 : 2.60  Gb         | 9  ->  8 : 4.10  Gb
link n°16 || 9  -> 10 : 1.40  Gb         | 10 ->  9 : 1.60  Gb
```

Figura 5.1: Resultado da tarefa 4.

Primeiramente temos os caminhos dos fluxos para a melhor solução encontrada.

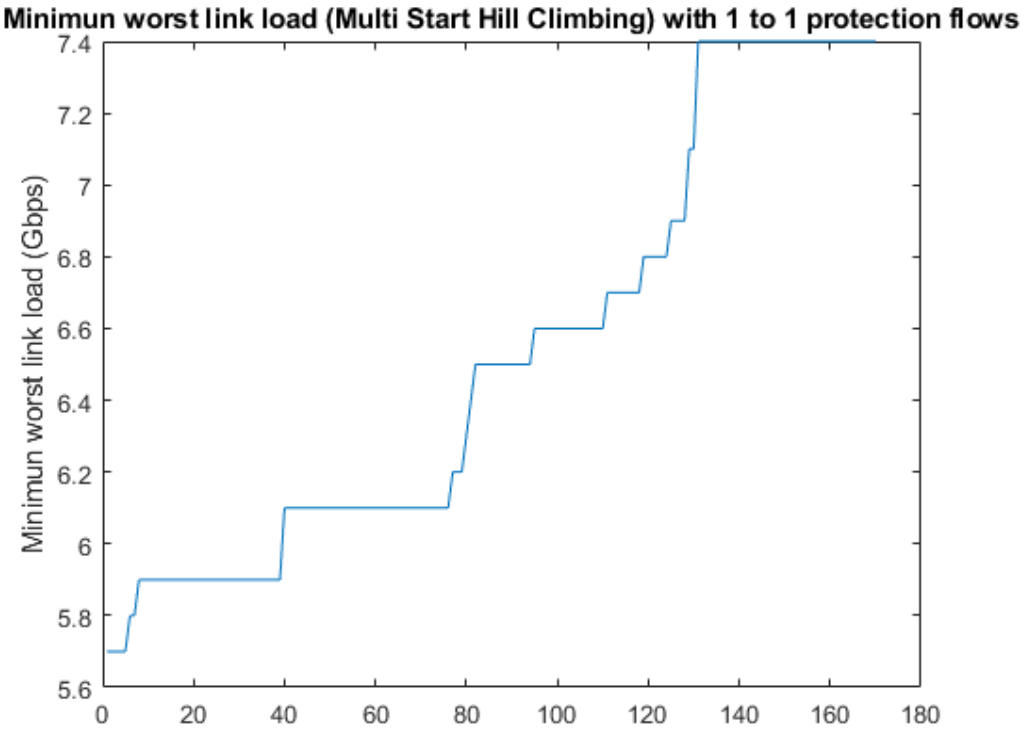

Figura 5.2: Gráfico resultante da execução da tarefa 4.

O que difere este exercício do anterior é que ao invés de calcular a protecção 1:1 com o caminho com a melhor *availability* e o seu par disjunto com a melhor *availability* também, neste a protecção 1:1 é calculada com com a melhor combinação entre os conjuntos de 10 melhores caminhos com maior *availability* e os seus pares de forma a minimizar a maior largura de banda num link.

Assim podemos observar, como esperado que a largura banda máxima suportada na rede neste exercício, 5.90 Gb, é muito menor para a melhor solução encontrada, no exercício 3.d), que foi 9.60 Gb (*link* número 10 na direção 7 -> 5). Por outro lado, o link com a menor largura de banda neste exercício foi o *link* nº6 com largura de 1.20 Gb e no exercício 3.d) também 1,20 Gb dada pelo link nº6. Sendo que na mesma rede, os mesmos fluxos foram considerados nos exercícios para a mesma quantidade de tráfego a diferença entre o máximo e o mínimo neste exercício e no 3.d) são, respectivamente, 5.90 - 1.20 = 4.70 Gb e 9.60 - 1.20 Gb = 8.40 Gb. Podemos então concluir que o conjunto de caminhos encontrada por este algoritmo distribui o tráfego de forma mais uniforme entre todos os *links* da rede do que no 3.d).

Tudo supracitado justifica ainda o porquê da soma da largura de banda dos links nas duas direcções ser maior. Quando a carga é distribuída de forma mais uniforme tenderá a haver mais fluxos em todas os *links* e o mecanismo de protecção 1:1 em cada *link* tem que garantir que tem largura de banda suficiente para o fluxo com a maior largura de banda que transita nesse link.

Comparando os valores de *availability* deste exercício com os obtidos no exercício 3.b), 4.2, podemos observar que estes são consistentemente piores, o que era esperado, com fluxos mais distribuídos de forma a minimizar a maior largura de banda num link com a protecção 1:1 é também esperado fluxos com caminhos mais longos. Sabendo que a *availability* de um fluxo é dada pela multiplicação da *availability* entre todos os links do fluxo. Caminhos mais longos(em número de hops) levam inevitavelmente a *availabilitys* mais baixas ou iguais.