Mini-Project #2

Traffic Engineering of Telecommunication Networks

Martim Neves, 88904
João Simões, 88930

9 de janeiro de 2023

Departamento de Eletrónica, Telecomunicações e Informática

Modelação e Desempenho de Redes e Serviços

Professor: Amaro Fernandes de Sousa

Contents

Task 1

Task 1.a.

Código MATLAB

```
load('InputDataProject2.mat')
nNodes= size(Nodes,1);
nFlows= size(T_uni,1);

k=1;
sPu= cell(1,nNodes);
sPa= cell(1,nNodes);
aNodes= [5,12];  % Anycast Nodes
T_any = [T_any(:, 1) zeros(length(T_any(:, 1)), 1) T_any(:, 2) T_any(:, 3)];
nodesTany=T_any(:, 1);

for f=1:nFlows
    [shortestPath, totalCost] = kShortestPath(L,T_uni(f,1),T_uni(f,2),k);
    sPu{f}= shortestPath;
    nSPu(f)= totalCost;
end

for n = 1:nNodes
    best = inf;

    if ~ismember(n, nodesTany)    % if the node does not have an anycastFlow skip it

        continue;
    end

    for a = 1:length(aNodes)
```

```
        [shortestPath, totalCost] = kShortestPath(L, n, aNodes(a), k);

        if totalCost < best
            sPa{n}(1)= shortestPath;
            best = totalCost;
            nSPa(n) = length(totalCost);
        end
    end
end

idx=1;
for p = sPa
    if isempty(p{1})
        continue;
    end
    T_any(idx, 2) = p{1}{1}(end);
    idx = idx + 1;
end
sPa = sPa(~cellfun(@isempty, sPa));

T=[T_uni;T_any];
sP=cat(2,sPu,sPa);

nAnyFlows= size(T_any,1);
sol= ones(1,nFlows+nAnyFlows);
sP = sP(~cellfun(@isempty, sP));
Loads = calculateLinkLoads(nNodes,Links,T,sP,sol);
maxLoad= max(max(Loads(:,3:4)));
fprintf('Worst link load = %.1f Gbps\n', maxLoad);
for i = 1:length(Loads)
    fprintf("{%d-%d}:  \t%.2f %.2f\n", Loads(i, 1),Loads(i, 2),Loads(i, 4),Loads(i, 3));
end
```

Análise do Código

Inicialmente são carregadas todas as matrizes dadas e delas são extraídos alguns valores necessários para a resolução do problema. Em seguida são declarados arrays para guardar valores de retorno, é criada uma matriz para armazenar os nós anycast destino da rede e outra matriz que guarda todos os nós source dos fluxos anycast. É também modificada a matriz que contém os fluxos anycast de maneira a acrescentar uma coluna de zeros, para mais tarde ser calculado o destino de cada fluxo.

O próximo passo é usar um ciclo *for* para percorrer todos os fluxos anycast e calcular o caminho mais curto (desde a origem até ao destino) para cada fluxo, guardando esse caminho e o custo associado. Logo depois é feito outro ciclo *for*, desta vez para percorrer todos os nós. Dentro deste ciclo é verificado se cada nó é um nó origem de um fluxo anycast e, se não for, é passado à frente. No entanto, se for origem de um fluxo anycast, é calculado o caminho mais curto desde esse nó até ambos os nós anycast destino, sendo que depois disso é escolhido o nó destino cujo caminho tem um custo menor, armazenando em variáveis esse caminho e o respetivo custo. Para terminar esta parte, os zeros introduzidos anteriormente na matriz dos fluxos anycast são agora substituídos pelo último nó no caminho mais curto de cada fluxo, ou seja, é alterada a matriz dos fluxos anycast de maneira a introduzir o nó destino para cada fluxo.

Por fim é feita a união de ambas as matrizes de fluxos numa só matriz e é feita a concatenação dos caminhos mais curtos para cada tipo de fluxo. É criada uma matriz com tamanho igual ao número de fluxos total existente e é invocada a função *calculateLinkLoads* para calcular a carga em cada ligação. Encontrando o valor máximo dessa matriz, encontramos assim o Worst Link Load.

Resultados

```
Worst link load = 49.9 Gbps
{1-2}:          8.60 10.60
{1-5}:          20.80 10.30
{1-7}:          5.60 3.40
{2-3}:          11.70 11.20
{2-4}:          13.10 7.30
{3-4}:          49.60 49.20
{3-6}:          21.00 19.80
{4-5}:          42.70 40.60
{4-8}:          49.20 33.10
{4-9}:          13.50 12.20
{5-7}:          10.00 14.70
{6-8}:          0.00 0.00
{6-14}:         14.40 7.60
{7-9}:          29.20 30.50
{8-11}:         15.20 20.20
{8-12}:         49.90 15.70
{9-10}:         28.30 28.90
{10-11}:        19.40 19.30
{11-13}:        19.40 19.30
{12-13}:        5.70 10.90
{12-14}:        7.10 21.30
{13-14}:        25.60 27.10
```

Análises e Justificações

Com estes resultados, conseguimos determinar qual a carga total que passa por cada uma destas ligações e retirar algumas conclusões da interpretação desses valores.

Uma dessas conclusões é que colocar todos os links com capacidade de 50 Gbps não é a solução ideal. Isto porque apenas existem 6 casos onde as ligações têm valores de carga entre 40 e 50 Gbps. O que significa que, para todos os restantes casos, a capacidade de 50 Gbps é excessiva, uma vez que não se está a usar 20% ou mais da capacidade máxima que a ligação permite. Assim, poderia ser delineada uma melhor gestão das capacidades das ligações, atribuindo uma menor capacidade máxima para os links utilizados por menos fluxos. Isto permitira não só poupar energia, mas também possivelmente reduzir custos de implementação das infraestruturas da rede.

Naturalmente, o Worst Link Load confere ser o valor máximo da matriz `Loads`, que neste caso corresponde à carga do fluxo entre os nós {8-12}: 49.90 Gbps.

Sabendo isto, outra conclusão que se pode retirar prende-se com o facto de um grande número de fluxos estar a preferir ser encaminhado através desta ligação, o que faz sentido, uma vez que são nós centrais nesta rede. No entanto, existem outros caminhos com cargas totais de ligação mais baixas que poderiam ser utilizadas em alternativa, de forma a não sobrecarregar tanto este fluxo {8-12}. Isto poderá mostrar-nos também que, caso a capacidade total dos links não estivesse limitada a 50 Gbps, este link muito provavelmente viria a ter uma carga total ainda maior, superior a 50.

Task 1.b.

Código MATLAB

```
idx=1;
sleepNodes=[];
sleepingNodes = '';
for i = 1 : length(Loads)
    if max(Loads(i, 3:4)) == 0
        sleepingNodes = append(sleepingNodes, ' {', num2str(Loads(i,1)), ',', num2str(Loads(i,2)), '}');
        aux=[Loads(i,1) Loads(i,2)];
        sleepNodes=[sleepNodes;aux];
        idx=idx+1;
    end
end
fprintf('List of links in sleeping mode:%s\n', sleepingNodes);

C=500;
t=zeros(1,nNodes);
for j=1:size(T,1)
    for k=1:length(sP{j}{1})
        t(sP{j}{1}(k))=t(sP{j}{1}(k))+T(j,3)+T(j,4);
    end
end
En=0;
for i=1:length(t)
    En=En+(10+90*(t(i)/C)^2);
end

El=0;
for i=1:length(L)-1
    for j=i+1:length(L)
        if(L(i,j)~=0 & L(i,j)~=Inf)
            if ~ismember([i j], sleepNodes,'rows')
                El=El+(6+0.2*L(i,j));
            else
                El=El+2;
            end
        end
    end
end
fprintf('The network energy consumption is %.4f\n', En+El);
```

```
fprintf('The nodes energy consumption is %.4f\n', En);
fprintf('The links energy consumption is %.4f\n',El);
```

Análise do Código

Inicialmente é verificado se existe alguma ligação na qual a carga seja zero em ambas as direções. Caso exista, os nós que formam essa ligação são adicionados a uma matriz de modo a formar um par. Em seguida é criada uma matriz para guardar a carga total em cada nó e são usados dois ciclos *for*, o primeiro para percorrer a matriz de fluxos e o segundo para percorrer o caminho mais curto para cada fluxo. Dessa forma, em cada nó que pertença ao caminho mais curto de cada fluxo, é possível somar as cargas em ambos os sentidos, de maneira a obter a carga total em cada nó. Tendo a carga total em cada nó é possível aplicar a fórmula para calcular a energia em cada nó e é feita a soma da energia em cada nó de maneira a obter a energia de todos os nós.

Para o cálculo da energia das ligações foram também usados dois ciclos *for*, um para percorrer a matriz com as distâncias de cada ligação até à penúltima posição e outro para percorrer essa mesma matriz até à última posição, sendo que este índice está sempre adiantado em relação ao índice do ciclo *for* anterior. Podendo assim aceder a cada posição desta matriz, se o valor em cada posição for diferente de zero e de infinito e se a ligação não estiver adormecida, é aplicada a fórmula para o cálculo da energia das ligações. Caso a ligação esteja adormecida, apenas é somada uma constante. Tendo agora a energia das ligações e a energia dos nós, basta somar as energias para obter o valor de consumo de energia da rede.

Resultados

```
List of links in sleeping mode: {6,8}
The network energy consumption is 851.8104
The nodes energy consumption is 188.2104
The links energy consumption is 663.6000
```

Task 1.c.

Código GreedyRandomized

```matlab
function [sol, load, Loads] = greedyRandomized(nNodes, Links, T, sP, nSP)
    nFlows= size(T,1);
    % random order of flows
    randFlows = randperm(nFlows);
    sol = zeros(1, nFlows);

    % iterate through each flow
    for flow = randFlows
        path_index = 0;
        best_load = inf;
        best_Loads = inf;

        % test every path "possible" in a certain load
        for path = 1 : nSP(flow)
            % try the path for that flow
            sol(flow) = path;
            % calculate loads
            Loads= calculateLinkLoads(nNodes,Links,T,sP,sol);
            load= max(max(Loads(:,3:4)));

            % check if the current load is better then bestLoad
            if load < best_load
                % change index of path and load
                path_index = path;
                best_load = load;
                best_Loads = Loads;
            end
        end
        sol(flow) = path_index;
    end
    load = best_load;
    Loads = best_Loads;
end
```

Análise do Código

Esta função percorre todos os fluxos aleatoriamente e, para cada caminho possível para cada fluxo, vai ver qual é o caminho que permite reduzir a carga máxima na rede. Para tal usa a função *calculateLinkLoads* e, em seguida, calcula a carga máxima proveniente da chamada dessa função. Por fim, se a carga máxima obtida for menor que a carga máxima atual, essa passa a ser a nova carga máxima e, quando todos os caminhos para todos os fluxos são verificados, a função devolve uma matriz que indica o melhor caminho para cada fluxo, a menor carga máxima encontrada e outra matriz com as cargas de todos os fluxos quando são usados os caminhos que minimizam a carga máxima.

Código HillClimbing

```
function [sol, load, Loads] = hillClimbing(nNodes, Links, T, sP, nSP, sol, load, Loads)
    nFlows= size(T,1);
    % set the best local variables
    bestLocalLoad = load;
    bestLocalSol = sol;
    bestLocalLoads = Loads;

    % Hill Climbing Strategy
    improved = true;
    while improved
        % test each flow
        for flow = 1 : nFlows
            % test each path of the flow
            for path = 1 : nSP(flow)
                if path ~= sol(flow)
                    % change the path for that flow
                    auxSol = sol;
                    auxSol(flow) = path;
                    % calculate loads
                    Loads = calculateLinkLoads(nNodes, Links,T,sP,sol);
                    auxLoad = max(max(Loads(:, 3:4)));

                    % check if the current load is better than start load
                    if auxLoad < bestLocalLoad
                        bestLocalLoad = auxLoad;
                        bestLocalSol = auxSol;
                        bestLocalLoads = Loads;
                    end
                end
```

```
            end
        end

        if bestLocalLoad < load
            load = bestLocalLoad;
            sol = bestLocalSol;
            Loads = bestLocalLoads;
        else
            improved = false;
        end
    end
end
```

Análise do Código

O algoritmo Hill Climbing volta a testar cada caminho de cada fluxo mas desta vez sem ser de forma aleatória. Tendo como base uma solução inicial proveniente do algoritmo *GreedyRandomized*, este algoritmo vai testar cada caminho possível para cada fluxo, de modo a otimizar a solução. Isto é feito até que deixem de ser encontradas melhorias na solução final, sendo que o valor guardado é o mais otimizado possível.

Task 1.d.

Código MATLAB

```
k=2;
sPud= cell(1,nFlows);
nSPud= zeros(1,nFlows);

for f=1:nFlows
    [shortestPath, totalCost] = kShortestPath(L,T_uni(f
        ,1),T_uni(f,2),k);
    sPud{f}= shortestPath;
    nSPud(f)= length(totalCost);
end

sPd=cat(2,sPud,sPa);
sPd = sPd(~cellfun(@isempty, sPd));
nSPd=cat(2,nSPud,nSPa);
nSPd=nonzeros(nSPd);
nSPd=nSPd';

% Optimization algorithm based on Multi Start Hill
    Climbing algorithm with Greedy Randomized strategy:
t = tic;
timeLimit= 30;

bestLoad = inf;
bestTime = 0;
contador= 0;
somador= 0;

while toc(t) < timeLimit

    [sol, load, Loads] = greedyRandomized(nNodes, Links,
        T, sPd, nSPd);

    [sol, load, Loads] = hillClimbing(nNodes, Links, T,
        sPd, nSPd, sol, load, Loads);

    if load<bestLoad
        bestSol= sol;
        bestLoad= load;
        bestLoads= Loads;
        bestTime = toc(t);
    end
    contador= contador+1;
```

```
    somador= somador+load;
end


idx=1;
sleepNodes=[];
for i = 1 : length(Loads)
    if max(Loads(i, 3:4)) == 0
        aux=[Loads(i,1) Loads(i,2)];
        sleepNodes=[sleepNodes;aux];
        idx=idx+1;
    end
end

C=500;
t=zeros(1,nNodes);
for j=1:size(T,1)
    for l=1:length(sPd{j}{bestSol(j)})
        t(sPd{j}{bestSol(j)}(l))=t(sPd{j}{bestSol(j)}(l))+T(j,3)+T(j,4);
    end
end
En=0;
for i=1:length(t)
    En=En+(10+90*(t(i)/C)^2);
end

El=0;
for i=1:length(L)-1
    for j=i+1:length(L)
        if(L(i,j)~=0 & L(i,j)~=Inf)

            if isempty(sleepNodes)
                El=El+(6+0.2*L(i,j));
            else
                if ~ismember([i j], sleepNodes, 'rows')
                    El=El+(6+0.2*L(i,j));
                else
                    El=El+2;
                end
            end

        end
    end
end
fprintf('Greedy Randomized - Hill Climbing:\n')
```

```
fprintf('W = %.2f Gbps, E = %.4f, time = %.2f sec\n',
    bestLoad, En+El, bestTime)
fprintf('No. of generated solutions = %d\n',contador);
fprintf('Avg. worst link load among all solutions= %.2f\n
    ',somador/contador);
```

Análise do Código

Inicialmente são calculados os shortest paths para os nós unicast mas, desta vez, são calculados dois shortest paths, em vez de apenas um. De seguida são unidos os shortest paths e os arrays que contêm o número de caminhos mais curtos para cada tipo de fluxo, calculados anteriormente. O passo seguinte é declarar o tempo limite para a corrida, inicializar variáveis para guardar os valores da simulação e, usando um ciclo *while* que tem a duração do tempo limite, invocar as funções externas explicadas anteriormente para calcular a melhor a carga máxima possível.

Acabado este ciclo, e percorrendo a matriz que armazena a carga em cada ligação, podemos verificar se existem ligações com zero de carga e, caso existam, essas ligações podem ser adormecidas. Por último são calculadas as energias dos nós e das ligações. Para a energia dos nós é percorrida a matriz de fluxos e, para cada fluxo, são somadas as cargas nos dois sentidos em todos os nós pertencentes ao caminho mais curto, de modo a calcular a carga total em cada nó. Tendo a carga total em cada nó é aplicada a fórmula para o cálculo da energia dos nós. Já para a energia das ligações, é percorrida a matriz com as distâncias entre os nós que formam uma ligação e é usada essa distância, assim como a fórmula dada, de modo a calcular a energia das ligações. Somando as duas energias, obtemos a energia total consumida pela rede.

Resultados

```
W = 40.60 Gbps, E = 896.0870, time = 0.02 sec
No. of generated solutions = 51239
Avg. worst link load among all solutions= 47.19
```

Análises e Justificações

Ao utilizar algoritmos de otimização com o objetivo de minimizar o worst link load, é expectável que o valor de W (worst link load) diminua, uma vez que se concentra em encontrar uma solução de encaminhamento que distribui os fluxos de forma mais uniforme através da rede, evitando assim sobrecarregar tanto cada um dos links.

Podemos comprovar isto, uma vez que o atual valor de W (40.60 Gbps) é inferior ao anterior valor de W calculado na tarefa 1.a (49.90 Gbps).

No entanto, existe um trade-off. Ao distribuir os fluxos de forma mais uniforme pela rede, os algoritmos são capazes de reduzir o worst link load mas, para compensar, poderá aumentar o consumo energético da rede. Isto pode ocorrer se o algoritmo de otimização encaminhar mais fluxos por caminhos mais longos, caminhos estes que requerem mais energia para transmitir, podendo até ser necessário reativar alguns nós que estavam adormecidos previamente. Resumindo, se o algoritmo de otimização aumentar a carga de algumas ligações a fim de reduzir o worst link load, isso pode levar ao aumento do consumo de energia da rede.

Podemos comprovar isto, uma vez que o atual valor de E (895.4257) é superior ao anterior valor de E calculado na tarefa 1.b (851.8104).

Task 1.e.

Código MATLAB

Para o código desta alínea, apenas alterámos o valor de k de 2 para 6 no código apresentado na alínea anterior.

Resultados

```
W = 40.60 Gbps, E = 896.0532, time = 0.08 sec
No. of generated solutions = 12455
Avg. worst link load among all solutions= 47.40
```

Análises e Justificações

Comparativamente aos resultados da alínea anterior, o valor do worst link load é exatamente igual e o valor da energia consumida pela rede é praticamente igual em ambos os casos, com algumas variações entre cada execução do programa.

Analisando o código, isto deve-se ao facto de o algoritmo Greedy Randomized priorizar sempre a escolha dos caminhos que apresentam o menor/melhor link load. Logo ao aumentarmos o k, isto é, o número de caminhos mais curtos calculados, não estamos necessariamente a introduzir melhores nem piores soluções, uma vez que este algoritmo escolhe sempre a solução com o menor W de entre todas e descarta as restantes. Aumentar o número de caminhos mais curtos calculados apenas implica uma melhor precisão de escolha do algoritmo. Mas, caso a solução já seja a melhor (como é no caso com k=2 *), então aumentar o valor de k não vai fazer qualquer diferença.

No entanto, alterar o k provocou sim uma diminuição considerável no número de soluções geradas, tendo sido encontradas agora apenas 12455, comparativamente com as 51239 na tarefa 1.d para k = 2. Isto pode ser explicado devido à função `kShortestPath()` ter de encontrar mais caminhos mais curtos adicionais, o que vai resultar num tempo de computação mais longo. Uma vez que o tempo limite não se alterou (timeLimit = 30) desde a tarefa anterior, foi necessário calcular mais 4 k-shortest paths adicionais com o mesmo tempo máximo, acabando por não haver tempo para calcular tantas soluções como na tarefa anterior.

Daí o tempo que demorou até a melhor solução ser encontrada ter aumentado e a média dos worst link loads de todas as soluções no geral ter piorado ligeiramente.

* Verificámos que esta solução já era a melhor possível ao testarmos o mesmo programa com um valor de k ainda maior (k = 10 e k = 20) e o valor de W manteve-se sempre o mesmo.

Task 2

Task 2.a.

Código calculateLinkEnergy

```
function [load, Loads, energy] = calculateLinkEnergy(nNodes,
    Links,T,sP,Solution, L)
    nFlows= size(T,1);
    nLinks= size(Links,1);
    aux= zeros(nNodes);
    for i= 1:nFlows
        if Solution(i)>0
            path= sP{i}{Solution(i)};
            for j=2:length(path)
                aux(path(j-1),path(j))= aux(path(j-1),path(j))
                    + T(i,3);
                aux(path(j),path(j-1))= aux(path(j),path(j-1))
                    + T(i,4);
            end
        end
    end
    Loads= [Links zeros(nLinks,2)];
    for i= 1:nLinks
       Loads(i,3)= aux(Loads(i,1),Loads(i,2));
       Loads(i,4)= aux(Loads(i,2),Loads(i,1));
    end
    load = max(max(Loads(:, 3:4)));
    if load > 50   % If the worst link load is greater than
        max capacity , energy will be infinite
        energy = inf;
    else
        energy = 0;
        for i= 1:nLinks
            % link in sleeping mode
            if max(Loads(i, 3:4)) == 0
                energy = energy + 2;
            else
                len = L(Loads(i, 1), Loads(i, 2));
```

```
                energy = energy + (6 + 0.2 * len);
            end
        end

        C=500;
        t=zeros(1,nNodes);
        for j=1:size(T,1)
            if Solution(j)>0
                for k=1:length(sP{j}{Solution(j)})
                    t(sP{j}{Solution(j)}(k))=t(sP{j}{Solution(
                        j)}(k))+T(j,3)+T(j,4);
                end
            end
        end
        En=0;
        for i=1:length(t)
            En=En+10+90*(t(i)/C)^2;
        end
        energy=energy+En;
    end
end
```

Análise do Código

O código para esta função foi desenvolvido com base no código da função *calculateLinkLoads*, que já era fornecido. Inicialmente são percorridos todos os fluxos e, caso haja um caminho para esse fluxo, esse caminho é percorrido e são somadas as cargas em ambos os sentidos, de modo a calcular a carga total nessa ligação. Caso a carga máxima na ligação seja superior à sua capacidade, que neste caso é 50, a energia na ligação é infinita. Caso contrário, é usado o mesmo código desenvolvido para a alínea 1.b de modo a calcular a energia, tanto nos nós, como nas ligações.

Código GreedyRandomizedEnergy

```
function [sol, load, Loads, energy] = greedyRandomizedEnergy(
    nNodes, Links, T, sP, nSP, L)
    nFlows = size(T,1);
    % random order of flows
    randFlows = randperm(nFlows);
    sol = zeros(1, nFlows);

    % iterate through each flow
    for flow = randFlows
        path_index = 0;
        best_load = inf;
        best_Loads = inf;
```

```
        best_energy = inf;

        % test every path "possible" in a certain load
        for path = 1 : nSP(flow)
            % try the path for that flow
            sol(flow) = path;
            % calculate loads
            [load, Loads, energy] = calculateLinkEnergy(nNodes,Links,T,sP,sol, L);

            % check if the current energy is better then best_energy
            if energy < best_energy
                % change index of path and load
                path_index = path;
                best_load = load;
                best_Loads = Loads;
                best_energy = energy;
            end
        end

        if path_index == 0
            break;
        else
            sol(flow) = path_index;
        end


    end
    load = best_load;
    Loads = best_Loads;
    energy = best_energy;
end
```

Análise do Código

O código para esta função foi desenvolvido com base no algoritmo *GreedyRandomized* usado na tarefa anterior, mas com as devidas adaptações de modo a otimizar a energia. A primeira diferença é a função invocada que, desta vez, calcula também a energia, ao invés de apenas calcular as cargas. A diferença seguinte está na comparação: enquanto que na tarefa anterior, o termo de comparação era a carga (uma vez que era esse o parâmetro de otimização), desta vez passa a ser a energia. Como tal, se a energia calculada através da chamada à função for menor que a energia atual, essa passa a ser a energia atual. A terceira e última diferença está na atribuição do melhor caminho. Desta vez, caso não haja nenhum caminho que minimize a energia, a função retorna sem valores.

Código HillClimbingEnergy

```
function [sol, load, Loads, energy] = hillClimbingEnergy(
    nNodes, Links, T, sP, nSP, sol, load, Loads, energy, L)
    nFlows= size(T,1);
    % set the best local variables
    bestLocalLoad = load;
    bestLocalSol = sol;
    bestLocalLoads = Loads;
    bestLocalEnergy = energy;

    % Hill Climbing Strategy
    improved = true;
    while improved
        % test each flow
        for flow = 1 : nFlows
            % test each path of the flow
            for path = 1 : nSP(flow)
                if path ~= sol(flow)
                    % change the path for that flow
                    auxSol = sol;
                    auxSol(flow) = path;
                    % calculate loads
                    [auxLoad, auxLoads, auxEnergy] =
                        calculateLinkEnergy(nNodes,Links,T,sP,
                        auxSol, L);

                    % check if the current load is better then
                        start load
                    if auxEnergy < bestLocalEnergy
                        bestLocalLoad = auxLoad;
                        bestLocalSol = auxSol;
                        bestLocalLoads = auxLoads;
                        bestLocalEnergy = auxEnergy;
                    end
                end
            end
        end

        if bestLocalEnergy < energy
            load = bestLocalLoad;
            sol = bestLocalSol;
            Loads = bestLocalLoads;
            energy = bestLocalEnergy;
        else
            improved = false;
        end
```

```
    end
end
```

Análise do Código

O código deste algoritmo apresenta muito poucas diferenças em relação ao usado na tarefa anterior, sendo que a única coisa que muda é que as comparações são feitas com a energia, em vez de com a carga, uma vez que nesta alínea o parâmetro de otimização é a energia.

Task 2.b.

Código MATLAB

```
load('InputDataProject2.mat')
nNodes= size(Nodes,1);
nLinks= size(Links,1);
nFlows= size(T_uni,1);
nAnyFlows= size(T_any,1);
aNodes= [5,12];  % Anycast Nodes
T_any = [T_any(:, 1) zeros(length(T_any(:, 1)), 1) T_any(:, 2)
     T_any(:, 3)];
nodesTany=T_any(:, 1);

% Computing up to k=2 shortest paths for all flows from 1 to
    nFlows:
k= 2;
sPu= cell(1,nFlows);
nSPu= zeros(1,nFlows);
for f=1:nFlows
    [shortestPath, totalCost] = kShortestPath(L,T_uni(f,1),
        T_uni(f,2),k);
    sPu{f}= shortestPath;
    nSPu(f)= length(totalCost);
end


sPa= cell(1,nAnyFlows);
nSPa= zeros(1,nAnyFlows);
for n = 1:nNodes
    best = inf;

    if ~ismember(n, nodesTany)    % if the node does not have
        an anycastFlow skip it

        continue;
    end

    for a = 1:length(aNodes)
        [shortestPath, totalCost] = kShortestPath(L, n, aNodes
            (a), 1);

        if totalCost < best
            sPa{n}(1)= shortestPath;
            best = totalCost;
            nSPa(n) = length(totalCost);
        end
    end
end

```

```
idx=1;
for p = sPa
    if isempty(p{1})
        continue;
    end
    T_any(idx, 2) = p{1}{1}(end);
    idx = idx + 1;
end
sPa = sPa(~cellfun(@isempty, sPa));

T=[T_uni;T_any];
sP=cat(2,sPu,sPa);
sP = sP(~cellfun(@isempty, sP));
nSP=cat(2,nSPu,nSPa);
nSP=nonzeros(nSP);
nSP=nSP';

t = tic;
timeLimit = 30;
bestEnergy = inf;
while toc(t) < timeLimit
    % greedy randomzied start
    % first greedy randomized solution
    [sol, load, Loads, energy] = greedyRandomizedEnergy(nNodes
        , Links, T, sP, nSP, L);
    while energy == inf
        [sol, load, Loads, energy] = greedyRandomizedEnergy(
            nNodes, Links, T, sP, nSP, L);
    end

    [sol, load, Loads, energy] = hillClimbingEnergy(nNodes,
        Links, T, sP, nSP, sol, load, Loads, energy, L);

    if energy < bestEnergy
        bestSol= sol;
        bestLoad= load;
        bestLoads = Loads;
        bestEnergy = energy;
        bestLoadTime = toc(t);
    end
end

fprintf('W = %.2f Gbps\tE = %.2f\ttime = %.2f\n', bestLoad,
    bestEnergy, bestLoadTime);
```

Análise do Código

Tal como na tarefa anterior, inicialmente são calculados os k caminhos mais curtos para os fluxos unicast, sendo que neste caso k toma o valor 2, e é calculado o destino para cada fluxo anycast, assim como o seu caminho mais curto. Depois disso são unidas as matrizes de fluxos e o array com os caminhos mais curtos, é declarado o tempo limite de cada corrida e são invocados os algoritmos de otimização, guardando localmente a melhor energia, a melhor carga e o melhor caminho. A principal diferença em relação à tarefa anterior, para além de estarmos a otimizar um parâmetro diferente, reside no ciclo *while* usado.

Este ciclo é necessário pois, tal como foi explicado anteriormente, caso a carga máxima seja maior que a capacidade da ligação, a energia é infinita. No entanto, devido à natureza aleatória do algoritmo *GreedyRandomized*, há soluções iniciais em que a carga máxima é maior que a capacidade da ligação, mas também há soluções em que é menor, sendo que nos casos em que é menor, a energia pode ser ainda mais otimizada. Assim sendo, o ciclo *while* garante que o algoritmo *GreedyRandomized* é corrido até encontrar pelo menos uma solução em que a energia não seja infinita, podendo esta ser melhorada mais tarde.

Resultados

```
W = 45.60 Gbps  E = 775.64      time = 0.14
```

Análises e Justificações

Nesta tarefa, utilizamos os mesmos algoritmos de otimização (Multi Start Hill Climbing e Greedy Randomized) mas desta vez com o objetivo de minimizar a energia consumida pela rede. Contrariamente ao que acontecia na tarefa 1, agora é expectável que o valor de E diminua, uma vez que o algoritmo de otimização tenta encaminhar o maior número de fluxos (tendo em conta a capacidade máxima dos links) por caminhos mais curtos que requerem menos energia para transmitir, permitindo assim a redução do consumo energético da rede.

Podemos comprovar isto, uma vez que o atual valor de E (775.64) é inferior ao anterior valor de E calculado na tarefa 1 (à volta de 895.4257), na qual priorizávamos a minimização do valor do worst link load.

Para compensar, existe também aqui um trade-off. Assim, o valor de W (worst link load) é esperado que aumente, uma vez que o algoritmo concentra-se em encaminhar mais fluxos através de uma única ligação para poupar energia, fazendo com que a carga que passa nessa ligação aumente.

Podemos comprovar isto, uma vez que o atual valor de W (45.60 Gbps) é superior ao anterior valor de W calculado na tarefa 1 (40.60 Gbps).

Task 2.c.

Código MATLAB

Para o código desta alínea, apenas alterámos o valor de k de 2 para 6 no código apresentado na alínea anterior.

Resultados

```
W = 48.50 Gbps  E = 701.06      time = 27.44
```

```
W = 49.90 Gbps  E = 701.00      time = 19.63
```

```
W = 48.00 Gbps  E = 700.65      time = 49.21
```

(nesta última execução foi alterado o tempo limite de 30 para 60 segundos)

Análises e Justificações

Como demonstramos através da presença das duas primeiras figuras, existem algumas variações entre cada execução deste programa. Ainda assim, comparativamente aos resultados da alínea anterior, o valor do worst link load foi sempre superior e o valor da energia consumida pela rede foi sempre inferior.

Um maior valor de k, isto é, um maior número de caminhos mais curtos calculados oferecem melhores resultados de energia (o critério que estamos a otimizar na tarefa 2), pelo facto de os algoritmos terem mais opções por onde escolher para encaminhar os fluxos do nó origem até ao nó destino correspondente. No entanto, isto requer uma demora maior tanto do algoritmo Greedy Randomized (que tem de verificar e escolher mais soluções válidas) como do algoritmo Hill Climbing (que tem de pesquisar pelos vizinhos de mais soluções devolvidas pelo Greedy), o que explica os tempos consideravelmente maiores. No entanto e como já abordámos na alínea anterior, esta diminuição da energia consumida pela rede tem de ser compensada com um aumento do worst link load, o que se confirma com os resultados que obtemos.

Quanto à variação dos resultados, esta pode ser explicada pelo tempo limite atribuído (30 segundos) não ser o suficiente para encontrar a melhor solução de todas. Suspeitámos que isto estaria a acontecer ao olharmos para o tempo em que a melhor solução, impressa no final da execução do programa, era encontrada. Observámos que os valores temporais eram notavelmente maiores comparativamente com todas as outras execuções dos algoritmos até agora, onde geralmente a solução era encontrada antes sequer de ter passado meio segundo desde o início da execução do programa.

Reparámos também que estes instantes temporais eram muito próximos do tempo limite, representando então que os 30 segundos de tempo limite era uma janela de tempo muito curta para a execução desta tarefa. Experimentámos então com um tempo limite de 60 segundos e confirmaram-se as nossas previsões: foi encontrada uma solução mais ideal para além dos 30 segundos de tempo limite definidos inicialmente (mais precisamente aos 49.21 segundos, como mostramos na terceira figura acima).

Tentámos ainda para tempos limites maiores (90 e 120 segundos), mas não obtivemos melhores valores. O que significa que 60 segundos é suficiente para o algoritmo encontrar a melhor otimização desta rede (W = 48.00 e E = 700.65 Gbps).

Task 2.d.

Análises e Justificações

Grande parte das diferenças entre as alíneas da tarefa 1 e as da tarefa 2 já foram referidas anteriormente nas várias alíneas antecedentes.

No entanto, a diferença mais imediata entre a tarefa 1.e e a tarefa 2.c é que o aumento do valor de k de 2 para 6 origina comportamentos diferentes na rede. Na tarefa 1.e, que, relembrando, prioriza a solução com o menor worst link load, aumentar o valor de k não ofereceu qualquer melhoria para a rede. Apenas ocorreu uma pequena variação na energia, a qual foi tão mínima que foi desprezada. Já na tarefa 2.c, a qual prioriza a solução com o valor mais baixo de energia consumida pela rede, o aumento de k foi crucial para uma melhoria significativa na poupança de energia. Porém não existem apenas aspetos positivos. Esta diminuição na energia consumida pela rede provocou um pequeno aumento do worst link load e um notável aumento do tempo até o algoritmo obter a sua melhor solução.

Em suma, existem compromissos entre a minimização do consumo de energia da rede e a minimização do worst link load. As vantagens e desvantagens de cada uma destas abordagens têm de ser previamente analisadas e estudadas e a decisão dependerá de características específicas de rede, como por exemplo, das capacidades disponíveis para cada link, dos comprimentos e taxas de consumo de energia dos links, da finalidade/objetivo da rede, do orçamento disponível para a implementação das infraestruturas da rede, etc. etc..

Task 3

Task 3.a.

Código calculateLinkEnergy3

```
function [load, Loads, energy] = calculateLinkEnergy3(nNodes, Links,T,sP,Solution, L)
    nFlows= size(T,1);
    nLinks= size(Links,1);
    aux= zeros(nNodes);
    for i= 1:nFlows
        if Solution(i)>0
            path= sP{i}{Solution(i)};
            for j=2:length(path)
                aux(path(j-1),path(j))= aux(path(j-1),path(j)) + T(i,3);
                aux(path(j),path(j-1))= aux(path(j),path(j-1)) + T(i,4);
            end
        end
    end
    Loads= [Links zeros(nLinks,2)];
    for i= 1:nLinks
       Loads(i,3)= aux(Loads(i,1),Loads(i,2));
       Loads(i,4)= aux(Loads(i,2),Loads(i,1));
    end
    load = max(max(Loads(:, 3:4)));
    if load > 100   % If the worst link load is greater than max capacity , energy will be infinite
        energy = inf;
    else
        energy = 0;
        for i= 1:nLinks
            % link in sleeping mode
            if max(Loads(i, 3:4)) == 0
                energy = energy + 2;
            elseif max(Loads(i,3)) > 50 | max(Loads(i,4)) > 50
                len = L(Loads(i, 1), Loads(i, 2));
```

```
                energy = energy + (8 + 0.3 * len);
            else
                len = L(Loads(i, 1), Loads(i, 2));
                energy = energy + (6 + 0.2 * len);
            end
        end

        C=500;
        t=zeros(1,nNodes);
        for j=1:size(T,1)
            if Solution(j)>0
                for k=1:length(sP{j}{Solution(j)})
                    t(sP{j}{Solution(j)}(k))=t(sP{j}{Solution(j)}(k))+T(j,3)+T(j,4);
                end
            end
        end
        En=0;
        for i=1:length(t)
            En=En+10+90*(t(i)/C)^2;
        end
        energy=energy+En;
    end
end
```

Análise do Código

O código desta função é praticamente igual ao código apresentado na tarefa anterior, sendo que as únicas diferenças são que a capacidade da ligação é mudada de 50 para 100 e é introduzida uma nova fórmula para o cálculo da energia das ligações, caso a ligação tenha uma capacidade de 100Gbps.

Código GreedyRandomized3

O código para este algoritmo é igual ao da tarefa anterior.

Código HillClimbing3

O código para este algoritmo é igual ao da tarefa anterior.

Task 3.b.

Código MATLAB

```
clear all
close all
clc

load('InputDataProject2.mat')
nNodes= size(Nodes,1);
nLinks= size(Links,1);
nFlows= size(T_uni,1);
nAnyFlows= size(T_any,1);
aNodes= [5,12];  % Anycast Nodes
T_any = [T_any(:, 1) zeros(length(T_any(:, 1)), 1) T_any(:, 2) T_any(:, 3)];
nodesTany=T_any(:, 1);

% Computing up to k=6 shortest paths for all flows from 1 to nFlows:
k= 6;
sPu= cell(1,nFlows);
nSPu= zeros(1,nFlows);
for f=1:nFlows
    [shortestPath, totalCost] = kShortestPath(L,T_uni(f,1),T_uni(f,2),k);
    sPu{f}= shortestPath;
    nSPu(f)= length(totalCost);
end


sPa= cell(1,nAnyFlows);
nSPa= zeros(1,nAnyFlows);
for n = 1:nNodes
    best = inf;

    if ~ismember(n, nodesTany)    % if the node does not have an anycastFlow skip it

        continue;
    end

    for a = 1:length(aNodes)
        [shortestPath, totalCost] = kShortestPath(L, n, aNodes(a), 1);

        if totalCost < best
            sPa{n}(1)= shortestPath;
            best = totalCost;
            nSPa(n) = length(totalCost);
```

```
        end
    end
end

idx=1;
for p = sPa
    if isempty(p{1})
        continue;
    end
    T_any(idx, 2) = p{1}{1}(end);
    idx = idx + 1;
end
sPa = sPa(~cellfun(@isempty, sPa));

T=[T_uni;T_any];
sP=cat(2,sPu,sPa);
sP = sP(~cellfun(@isempty, sP));
nSP=cat(2,nSPu,nSPa);
nSP=nonzeros(nSP);
nSP=nSP';

t = tic;
timeLimit = 60;
bestEnergy = inf;
while toc(t) < timeLimit
    % greedy randomzied start
    % first greedy randomized solution
    [sol, load, Loads, energy] = greedyRandomizedEnergy3(
        nNodes, Links, T, sP, nSP, L);
    while energy == inf
        [sol, load, Loads, energy] = greedyRandomizedEnergy3(
            nNodes, Links, T, sP, nSP, L);
    end

    [sol, load, Loads, energy] = hillClimbingEnergy3(nNodes,
        Links, T, sP, nSP, sol, load, Loads, energy, L);

    if energy < bestEnergy
        bestSol= sol;
        bestLoad= load;
        bestLoads = Loads;
        bestEnergy = energy;
        bestLoadTime = toc(t);
    end
end

idx=1;
changedLinks = '';
for i = 1 : length(bestLoads)
    if max(bestLoads(i,3)) > 50 | max(bestLoads(i,4)) > 50
```

```
        changedLinks = append(changedLinks, ' {', num2str(
            bestLoads(i,1)), ',', num2str(bestLoads(i,2)), '}'
            );
    end
end
fprintf('List of links that changed capacity to 100Gbps:%s\n',
     changedLinks);
fprintf('W = %.2f Gbps\tE = %.2f\ttime = %.2f\n', bestLoad,
    bestEnergy, bestLoadTime);
```

Análise do Código

Inicialmente foram calculados os 6 caminhos mais curtos para os fluxos unicast e também os nós destino e caminhos mais curtos para os fluxos anycast. Em seguida fez-se a concatenação das matrizes de fluxos e dos caminhos mais curtos de cada tipo de fluxo, definiu-se o tempo limite para a corrida e foram chamados os algoritmos de otimização de maneira a obter a configuração de rede que produz o menor valor de energia máxima, guardando esse valor e outros dados de relevo e verificando que ligações mudaram a sua capacidade para 100Gbps.

Resultados

```
List of links that changed capacity to 100Gbps: {3,4}
W = 80.30 Gbps   E = 678.80      time = 0.96
```

Análises e Justificações

Devido a darmos a possibilidade de alguns links aumentaram a capacidade de 50 para 100 Gbps, seria expectável que o valor de W (worst link load) aumentasse comparativamente com os resultados da tarefa 2.c, o que se verifica que acontece. Obtivemos apenas um caso nesta situação: o fluxo $\{3, 4\}$. Fluxo este que é precisamente o fluxo onde obtivemos o worst link load, uma vez que é o único que mudou a capacidade máxima de 50 para 100 Gbps, ficando capaz de aguentar 80.30 Gbps.

Havendo agora a possibilidade de existirem links com capacidade máxima de 100 Gbps e mesmo estes links tendo um consumo de energia naturalmente maior, verificou-se que o valor de energia conseguiu, ainda assim, diminuir. Ao conseguir transmitir mais dados através fluxo $\{3, 4\}$, a rede conseguiu colocar mais alguns nós em sleeping mode, economizando mais energia do que na tarefa 2.c.

O facto de termos aumentado o tempo limite para 60 segundos também ajudou o algoritmo a obter a solução ideal, uma vez que tem tempo suficiente para calcular todas as soluções possíveis para todos os k-shortest paths com k = 6.

Task 3.c.

Código MATLAB

```
load('InputDataProject2.mat')
nNodes= size(Nodes,1);
nLinks= size(Links,1);
nFlows= size(T_uni,1);
nAnyFlows= size(T_any,1);
nodesTany=T_any(:, 1);
anycastCandidates=[4 5 6 12 13];

node1=0;
node2=0;
globalWorstLink=0;
globalBestEnergy=Inf;
globalLinksChanged='';
for y=1:length(anycastCandidates)-1
    for z=y+1:length(anycastCandidates)

        % Computing up to k=2 shortest paths for all flows
            from 1 to nFlows:
        k= 6;
        sPu= cell(1,nFlows);
        nSPu= zeros(1,nFlows);
        for f=1:nFlows
            [shortestPath, totalCost] = kShortestPath(L,T_uni(
                f,1),T_uni(f,2),k);
            sPu{f}= shortestPath;
            nSPu(f)= length(totalCost);
        end



        T_any = [T_any(:, 1) zeros(length(T_any(:, 1)), 1)
            T_any(:, 2) T_any(:, 3)];
        aNodes= [anycastCandidates(y),anycastCandidates(z)];
            % Anycast Nodes

        sPa= cell(1,nAnyFlows);
        nSPa= zeros(1,nAnyFlows);
        for n = 1:nNodes
            best = inf;

            if ~ismember(n, nodesTany)    % if the node does
                not have an anycastFlow skip it

                continue;
            end

```

```
            for a = 1:length(aNodes)
                [shortestPath, totalCost] = kShortestPath(L, n
                    , aNodes(a), 1);

                if totalCost < best
                    sPa{n}(1)= shortestPath;
                    best = totalCost;
                    nSPa(n) = length(totalCost);
                end
            end
        end

        idx=1;
        for p = sPa
            if isempty(p{1})
                continue;
            end
            T_any(idx, 2) = p{1}{1}(end);
            idx = idx + 1;
        end
        sPa = sPa(~cellfun(@isempty, sPa));

        T=[T_uni;T_any];
        sP=cat(2,sPu,sPa);
        sP = sP(~cellfun(@isempty, sP));
        nSP=cat(2,nSPu,nSPa);
        nSP=nonzeros(nSP);
        nSP=nSP';

        t = tic;
        timeLimit = 60;
        bestEnergy = inf;
        while toc(t) < timeLimit
            % greedy randomzied start
            % first greedy randomized solution
            [sol, load, Loads, energy] =
                greedyRandomizedEnergy3(nNodes, Links, T, sP,
                nSP, L);
            while energy == inf
                [sol, load, Loads, energy] =
                    greedyRandomizedEnergy3(nNodes, Links, T,
                    sP, nSP, L);
            end

            [sol, load, Loads, energy] = hillClimbingEnergy3(
                nNodes, Links, T, sP, nSP, sol, load, Loads,
                energy, L);

            if energy < bestEnergy
                bestSol= sol;
```

```
                bestLoad= load;
                bestLoads = Loads;
                bestEnergy = energy;
                bestLoadTime = toc(t);
            end
        end

        idx=1;
        changedLinks = '';
        for i = 1 : length(bestLoads)
            if max(bestLoads(i,3)) > 50 | max(bestLoads(i,4)) > 50
                changedLinks = append(changedLinks, ' {', num2str(bestLoads(i,1)), ',', num2str(bestLoads(i,2)), '}');
            end
        end

        fprintf('aNodes = {%d , %d}, W = %.2f Gbps, E = %.2f, Links changed =%s\n',anycastCandidates(y), anycastCandidates(z), bestLoad, bestEnergy, changedLinks);

        if bestEnergy < globalBestEnergy
            globalBestEnergy = bestEnergy;
            globalWorstLink = bestLoad;
            node1 = anycastCandidates(y);
            node2 = anycastCandidates(z);
            globalLinksChanged = changedLinks;
        end
    end
end

fprintf('\nSelected anycast nodes: {%d , %d}\n', node1, node2);
fprintf('List of links that changed capacity to 100Gbps:%s\n', globalLinksChanged);
fprintf('W = %.2f Gbps\tE = %.2f', globalWorstLink, globalBestEnergy);
```

Análise do Código

O código para esta alínea é muito semelhante ao usado na alínea anterior com a exceção de que, desta vez, são usados dois ciclos *for* de modo a percorrer todas as combinações possíveis de nós anycast, para que possamos perceber qual o par que produz melhores resultados. Além disso, foram também introduzidas algumas variáveis para guardar os melhores resultados.

Resultados

```
aNodes = {4 , 5}, W = 98.00 Gbps, E = 693.20, Links changed = {3,4} {4,5} {4,8}
aNodes = {4 , 6}, W = 87.50 Gbps, E = 666.43, Links changed = {3,4}
aNodes = {4 , 12}, W = 70.60 Gbps, E = 679.60, Links changed = {3,4}
aNodes = {4 , 13}, W = 61.80 Gbps, E = 709.32, Links changed = {4,8}
aNodes = {5 , 6}, W = 59.30 Gbps, E = 740.33, Links changed = {3,4}
aNodes = {5 , 12}, W = 49.60 Gbps, E = 682.13, Links changed =
aNodes = {5 , 13}, W = 56.90 Gbps, E = 728.13, Links changed = {11,13}
aNodes = {6 , 12}, W = 75.60 Gbps, E = 728.84, Links changed = {3,4} {4,8} {8,12}
aNodes = {6 , 13}, W = 64.30 Gbps, E = 778.95, Links changed = {7,9} {11,13}
aNodes = {12 , 13}, W = 85.60 Gbps, E = 749.96, Links changed = {4,5} {4,8} {8,12} {11,13}

Selected anycast nodes: {4 , 6}
List of links that changed capacity to 100Gbps: {3,4}
W = 87.50 Gbps  E = 666.43
```

Análises e Justificações

Como observamos pela figura, o par de nós anycast que oferecem o melhor/mais baixo valor de energia consumida pela rede são os nós anycast 4 e 6, sendo ele E = 666.43. O que nos mostra que a escolha de nós anycast utilizada até agora (nós 5 e 12) não era a melhor configuração possível para esta rede.

De facto, o nó 4 faz sentido que seja um dos melhores candidatos, uma vez que os nós anycast devem estar situados em locais estratégicos, de modo a serem facilmente acedidos pelos respetivos links ativos e o nó 4 é precisamente um dos nós que se encontram no centro desta rede. Para além disso, uma vez que se espera que o serviço anycast suporte um grande volume de tráfego, também faz sentido que o nó 4 tenha sido escolhido para nó anycast, visto que tem uma capacidade máxima mais elevada em relação a todos os outros links (100 Gbps).

No entanto, ao fazermos uma análise mais detalhada, reparamos que a diferença da energia entre ambas as configurações não foi tão significativa assim, apenas houve uma redução de aproximadamente 1.82%. Naturalmente, devido a esta diminuição da energia, o valor de W sofreu um aumento. Só que este aumento já foi na casa dos 8.96%. Assim, e contrariamente ao que esperávamos para esta alínea, na nossa opinião não compensava realizar esta troca de Data Centers para hospedar o servidor deste serviço.

Quanto aos nós que aumentaram a sua capacidade para 100 Gbps, não esperávamos este comportamento. Esperávamos que a melhor combinação de nós anycast tirasse mais proveito desta melhoria e que mais links alterassem a sua capacidade máxima para 100 Gbps. No entanto, isso não aconteceu para este caso, alterando-se apenas o link $\{3, 4\}$, tal como acontecia na alínea anterior. Ainda assim aconteceu para outras combinações de nós anycast, como podemos observar na figura.