Universidade de Aveiro

Departamento de Eletrónica, Telecomunicações e Informática

universidade de aveiro

Modelação e Desempenho de Redes e Serviços

Projeto 2

Tiago Alves (104110)

Rafael Amorim (98197)

28 de dezembro de 2023

Índice

Índice das Figuras

Índice da Tabela

Introdução

De acordo com o solicitado no mini projeto 2 da unidade curricular de Modelação e Desempenho de Redes e Serviços realizou-se este relatório apresentando excertos de código importantes para a explicação do raciocínio e descrevendo de forma sintetizada todas as conclusões retiradas dos resultados de cada exercício.

A estrutura do relatório consiste em quatro partes, uma para cada tarefa.

O código do projeto, tal como, toda a gestão de tarefas encontra-se disponível em:

https://github.com/Tiago-AlvesUA /MDRS/

Tarefa 1

Exercício 1a

Não é possível utilizar a solução de encaminhar todo o tráfego pelo caminho com menor atraso de propagação da rede. Cada ligação tem uma capacidade máxima de 100 Gbps, logo, se todo o tráfego fosse encaminhado por um único caminho (caminho que minimiza o atraso de propagação da rede) o valor da capacidade máxima da ligação seria ultrapassado.

Exercício 1b

Código:

O código implementado nesta alínea foi retirado em grande parte do trabalho já realizado nos guiões. Primeiro é necessário definir uma quantidade, **k**, de caminhos mais curtos para cada fluxo, que serão calculados com a função *"kShortestPath"*. A estes caminhos será aplicado o algoritmo *"Multi Start Hill Climbing"*, que combina os algoritmos *"Greedy Randomized"* e *"Hill Climbing"*, obtendo como solução uma escolha dos caminhos otimizando o parâmetro pretendido.

```
%%% Computing up to k=2 shortest paths for all flows of service 1: %%%
k= 2;
sP= cell(1,nFlows);
nSP= zeros(1,nFlows);
for f=1:nFlows
    [shortestPath, totalCost] = kShortestPath(L,T(f,1),T(f,2),k);
    sP{f}= shortestPath;
    nSP(f)= length(totalCost);
end

t= tic;
timeLimit= 60; % runtime limit of 60 seconds
bestLoad= inf;
contador= 0;
somador= 0;
while toc(t) < timeLimit
    sol= zeros(1,nFlows);
    [sol, ~, linkEnergy] = Task1_GreedyRandomized(nNodes,Links,T,L,sP,nSP,sol);

    [sol, load, Loads, linkEnergy] = Task1_hillClimbing(sol,nNodes,Links,T,L,sP,n
Energy);

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

Figura 1: Código Task1b

Como o objetivo é minimizar a carga máxima nas ligações, é possível notar que a cada solução, para que esta seja aceite como melhor e habilitada para substituir a anterior, é verificado se a carga máxima ('load') é menor que a melhor carga máxima anteriormente obtida.

A função do algoritmo "*Greedy Randomized*" encontra-se presente abaixo:

```
function [sol,load,linkEnergy] = Task1_GreedyRandomized(nNodes,Links,T,L,sP,nSP,s
    nFlows = size(T,1);

    for flow= randperm(nFlows) % ordem random dos fluxos
        path_index = 0;
        best_load = inf;
        best_energy = inf;

        for path = 1 : nSP(flow)
            sol(flow) = path;
            [Loads,linkEnergy] = calculateLinkLoads(nNodes, Links, T,L, sP, sol);
            maxLoad = max(max(Loads(:,3:4)));

            if maxLoad < best_load % se load obtida for menor a melhor obtida ant
mente, trocar
                path_index = path;
                best_load = maxLoad;
                best_energy = linkEnergy;
            end
        end
        sol(flow) = path_index;
    end
    load = best_load;
    linkEnergy = best_energy;
end
```

Figura 2: Task1 Greedy Randomized

Para cada fluxo, numa ordem aleatória, são analisados todos os caminhos como possíveis soluções. Se a carga máxima obtida para a solução for menor que a anterior guardada, os resultados são guardados.

Em baixo podemos também observar o código do algoritmo "*Hill Climbing*":

```
function [sol,load,Loads,linkEnergy]= Task1_hillClimbing(sol,nNodes,Links,T,L,sP,
ergy)
    nFlows = size(T,1);
    Loads= calculateLinkLoads(nNodes,Links,T,L,sP,sol);
    load= max(max(Loads(:,3:4)));
    improved = true;
    linkEnergy = energy;
    while improved
        loadBestNeigh = inf;
        for flow= 1:nFlows
            for path= 1:nSP(flow) %Rodar por todos os vizinhos possiveis
                if sol(flow)~=path %Não trocar a sol por ela mesma
                    auxsol = sol;
                    auxsol(flow)= path;
                    [auxLoads,auxEnergy]= calculateLinkLoads(nNodes,Links,T,L,sP,
                    auxload= max(max(auxLoads(:,3:4)));
                    % check if the neighbour load is better than the
                    % current load
                    if auxload<loadBestNeigh
                        loadBestNeigh= auxload;
                        LoadsBestNeigh= auxLoads;
                        fbest = flow;
                        pbest = path;
                        energyBestNeigh= auxEnergy;
                    end
                end
            end
        end
        if loadBestNeigh<load % Encontrado melhor vizinho, trocar valores
            load= loadBestNeigh;
            sol(fbest)= pbest;
            Loads= LoadsBestNeigh;
            linkEnergy = energyBestNeigh;
        else
            improved = false;
        end
    end
end
```

Figura 3: Task1 Hill Climbing

Por cada fluxo vamos analisar todos os vizinhos. É feita uma cópia da solução, 'auxsol', e a partir desta conseguimos obter a carga deste vizinho. Se a carga resultante for melhor que a melhor carga anteriormente guardada, de outro vizinho, esta é guardada como nova melhor carga resultante.

Depois de percorrer todos os fluxos se a carga da vizinhança for melhor que a carga atual, 'load', esta solução é considerada melhor e é feita a atualização dos restantes valores.

Exercício 1c

Código:

Para alguns dos valores pedidos neste exercício, foram necessários alguns cálculos simples, enquanto que para outros a sua obtenção foi praticamente direta. A solução é escolhida de 2 caminhos mais curtos calculados previamente (k=2), num tempo limite de 60 segundos (timeLimit=60). A cada iteração, na ocorrência de mudança da melhor solução, para além do melhor/menor valor de carga máxima, foram também guardados os valores das seguintes soluções:

- Cargas para todas as ligações, denominadas 'bestLoads'
- Energia consumida pelas ligações, 'bestEnergy'
- Tempo despendido para encontrar a solução, 'bestTime'

Além disso é incrementado o valor do contador a cada ciclo de execução do algoritmo.

```
%%% Computing up to k=2 shortest paths for all flows of service 1: %%%
k= 2;
sP= cell(1,nFlows);
nSP= zeros(1,nFlows);
for f=1:nFlows
    [shortestPath, totalCost] = kShortestPath(L,T(f,1),T(f,2),k);
    sP{f}= shortestPath;
    nSP(f)= length(totalCost);
end

t= tic;
timeLimit= 60; % runtime limit of 60 seconds
bestLoad= inf;
contador= 0;
while toc(t) < timeLimit
    sol= zeros(1,nFlows);
    [sol, ~, linkEnergy] = Task1_GreedyRandomized(nNodes,Links,T,L,sP,nSP,sol);

    [sol, load, Loads, linkEnergy] = Task1_hillClimbing(sol,nNodes,Links,T,L,sP,nSP,link-
Energy);

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

Figura 4: Código Task1c

Pior carga e carga média das ligações:

Para obter a pior/maior carga de ligação da solução basta verificar o valor armazenado na variável 'bestLoad'. Quanto à carga média das ligações basta dividir a soma das cargas das ligações em 'bestLoads' pelo número total das ligações, e por 2 já que os valores de 'bestLoads' são de ida e volta.

Energia:

No que toca à energia, foram necessárias algumas alterações no cálculo de cargas em ambos os algoritmos e foi ainda necessário criar uma função que retorna a energia consumida por cada nó/router da rede.

O código seguinte foi adicionado à função "calculateLinkLoads":

```
function [Loads,solLinkEnergy]= calculateLinkLoads(nNodes,Links,T,L,sP,Solution)
    (...)
    for i= 1:nLinks
        Loads(i,3)= aux(Loads(i,1),Loads(i,2));
        Loads(i,4)= aux(Loads(i,2),Loads(i,1));

        % Energy calculation
        if (Loads(i,3:4) == 0) % Link is in sleeping mode
            energy = 2;
        else
           link_length = L(Loads(i,1),Loads(i,2));
           energy = 9 + 0.3 * link_length;
        end
        solLinkEnergy = solLinkEnergy + energy;
    end

    %If biggest link load is greater then link capacity energy is inf
    maxLoad = max(max(Loads(:,3:4)));
    if maxLoad > 100
        solLinkEnergy = inf;
    end
end
```

Figura 5: Cálculo da energia de ligação

Ao percorrer as ligações verificamos se a ligação não está a ser usada. Se não estiver, a ligação está em "sleeping mode" e a energia despendida pela mesma é 2. Caso contrário é obtido o comprimento entre os dois nós da ligação, utilizando a matriz L. Depois é aplicada a fórmula $9 + 0.3 * l$, onde l é o comprimento da ligação. Estes valores são incrementados à energia total da solução.

Uma nota importante para o exercício 2 do projeto é a última verificação. Caso o valor de carga máxima de ligação obtido ultrapasse a capacidade máxima das ligações que é 100 Gbps é atribuído o valor máximo de energia à solução. Assim esta solução será sempre descartada.

Quanto aos algoritmos, garantimos que o valor da energia da solução obtido usando a função acima também fosse guardado.

Por forma a obter a energia total da solução falta ainda calcular a energia consumida nos nós. Criámos, por isso, a função "calculateNodeEnergy":

```
function nodesEnergy = calculateNodeEnergy(T,sP,nNodes,Solution)
    nFlows= size(T,1);
    % criar o vetor para guardar valores do TT para cada nó
    nodes_TT = zeros(1,nNodes);
    for i= 1:nFlows
        if Solution(i)>0
            path= sP{i}{Solution(i)};
            for node = path
                % somar o TT para os nos todos aqui
                nodes_TT(node) = nodes_TT(node) + sum(T(i,3:4));
            end
        end
    end
    nodesEnergy = sum(20 + 80 * sqrt(nodes_TT/1000));
end
```

Figura 6: Cálculo da energia dos nós

Para cada fluxo verificamos qual o caminho escolhido na solução obtida. Dentro do caminho acumulamos, para cada router, o tráfego que é suportado por ele. Depois de obtidos estes valores, é usada a fórmula $20 + 80 * \sqrt{t}$, onde t representa o tráfego suportado por cada router a dividir pela sua capacidade máxima, de 1000 Gbps.

Na tarefa principal, depois de obtida a solução com os algoritmos, basta então calcular a energia dos nós para essa solução e somá-la à energia das ligações já calculada e guardada.

```
% Calculate energy comsuption
nodesEnergy = calculateNodeEnergy(T,sP,nNodes,bestSol);
total_energy = nodesEnergy + bestLinkEnergy;
```

Figura 7: Cálculo da energia total consumida

Média do atraso de propagação de ida e volta (de cada serviço)**:**

Para obter o atraso médio de propagação de ida e volta dos dois serviços criámos a seguinte função:

```
function [avgDelay1, avgDelay2] = calculateServiceDelays(sP, sol, D, nFlows1, nFlows2)
    delays1 = zeros(1, nFlows1);
    delays2 = zeros(1, nFlows2);
    % Calculate delays for service 1
    for f = 1:nFlows1
        if sol(f) ~= 0
            path = sP{f}{sol(f)};
            delays1(f) = 2 * sumPathDelays(path, D); % Round-trip delay
        else
            delays1(f) = inf; % Assign a large delay if no path exists
        end
    end
    avgDelay1 = mean(delays1);

    % Calculate delays for service 2
    % Make sure to start at 1 for delays2 indexing
    for f = 1:nFlows2
        flowIndex = f + nFlows1; % Adjust index for the second service
        if sol(flowIndex) ~= 0
            path = sP{flowIndex}{sol(flowIndex)};
            delays2(f) = 2 * sumPathDelays(path, D); % Round-trip delay
        else
            delays2(f) = inf; % Assign a large delay if no path exists
        end
    end
    avgDelay2 = mean(delays2);
end

function totalDelay = sumPathDelays(path, D)
    totalDelay = 0;
    for j = 2:length(path)
        totalDelay = totalDelay + D(path(j-1), path(j));
    end
end
```

Figura 8: Atrasos de propagação dos serviços

Primeiro percorremos todos os fluxos do serviço 1, e para cada um destes, obtemos o atraso de propagação para o respetivo caminho (soma dos atrasos de cada ligação no caminho) escolhido pela solução. Este valor é depois multiplicado por 2, por estarmos a calcular o valor do atraso de ida e volta.

Depois de percorrer todos os fluxos é feita a média dos atrasos de propagação.

Para percorrer os fluxos do próximo serviço, como estes se encontram seguidos na solução, começamos a iteração dos fluxos em 1 mais o comprimento dos fluxos anterior.

Número e lista de ligações que não suportam nenhum tráfego:

As ligações que não suportam nenhum tráfego são as que não têm tráfego tanto para um lado como para o outro. Portanto, se a soma das colunas 3 e 4 do ‘bestLoads’ for 0, essa ligação é uma das quais não está a suportar nenhum tráfego:

```
% Calculate links not supporting any traffic flow
linksNoTraffic = [];
for i=1:nLinks
    if sum(bestLoads(i,3:4)) == 0
        linksNoTraffic = [linksNoTraffic,i];
    end
end
```

Figura 9: Ligações sem tráfego

Número de ciclos e tempo de obtenção da melhor solução:

O número de ciclos é incrementado a cada ciclo da iteração, onde é executado o algoritmo. O tempo para obtenção da melhor solução corresponde ao valor 'bestTime', já referido acima.

Exercício 1d

Para o exercício 1 d) apenas foi necessário mudar o valor da variável k para 6, de forma que a melhor solução seja escolhida de entre 6 caminhos mais curtos.

Exercício 1e

Resultados:

```
Worst link load of the (best) solution = 87.80 Gbps
Average link load of the solution = 37.63 Gbps
Network energy comsuption of the solution = 2845.68
Average round-trip propagation delay of service 1 = 0.005637 sec
Average round-trip propagation delay of service 2 = 0.005611 sec
Number of links not supporting any traffic flow = 4
List of links not supporting any traffic flow:
    1 -> 5
    2 -> 5
    6 -> 15
    12 -> 13
Number of cycles run by the algorithm = 24709
Time obtained best solution= 0.041078 sec
```

Figura 10: Valores para 1 c), k = 2

```
Worst link load of the (best) solution = 68.60 Gbps
Average link load of the solution = 43.96 Gbps
Network energy comsuption of the solution = 3097.65
Average round-trip propagation delay of service 1 = 0.006994 sec
Average round-trip propagation delay of service 2 = 0.006391 sec
Number of links not supporting any traffic flow = 1
List of links not supporting any traffic flow:
    6 -> 15
Number of cycles run by the algorithm = 4688
Time obtained best solution= 7.222328 sec
```

Figura 11: Valores para 1 d), k = 6

Conclusões:

Começando por analisar os valores obtidos, podemos verificar que o valor da carga máxima de ligação da solução obtida desceu quando utilizamos k = 6 (6 caminhos mais curtos para cada fluxo, calculados à priori). Como a prioridade é obter uma solução que minimize a carga máxima, o fornecimento ao algoritmo de um conjunto maior de caminhos mais curtos para cada fluxo, permite uma maior variedade de escolha e assim encontrar um caminho cuja carga máxima é menor do que a obtida para a solução com k = 2.

Esta otimização do valor da carga máxima, não se traduz numa melhoria equivalente da carga média nas ligações. Isto deve-se ao facto de estarmos focados na procura de uma menor carga máxima e não na redução da carga média de todas as ligações, resultando em valores médios de cargas das ligações em k=6, eventualmente, maiores do que os obtidos em k=2.

Quando otimizamos um certo parâmetro podemos esperar uma diminuição na

performance dos restantes como é possível verificar com a energia. Com maior número de caminhos conseguimos diminuir a carga máxima, mas, consequentemente, a energia aumenta.

O tempo de atraso médio de propagação de ida e volta para ambos os serviços aumentou. A matriz com os atrasos de propagação entre cada ligação, D, é diretamente proporcional aos valores de comprimento de ligações, presentes na matriz L. A cada novo caminho mais curto calculado, utilizando a função "kShortestPath", obtemos um caminho mais longo, possivelmente até com mais nós e ligações que o anterior. Logo, utilizando k=6, alguns caminhos escolhidos na solução serão maiores, resultando em atrasos de propagação superiores.

O número de ligações que não suportam nenhum tráfego diminui substancialmente para os segundos valores. De acordo com a explicação das consequências do aumento do número de caminhos, explicado no parágrafo de cima, mais ligações poderão estar a ser usadas nos novos caminhos calculados. Como a solução é escolhida de entre qualquer um destes caminhos, o número de ligações inativas diminui.

O número de ciclos de corridos pelo algoritmo diminui porque a solução passa a ter de ser obtida a partir de um número maior de caminhos possíveis. No algoritmo "Greedy Randomized" os fluxos e os seus respetivos caminhos são percorridos um a um e no "Hill Climbing" também teremos um número maior de vizinhos. Consequentemente, o tempo para percorrer o algoritmo aumentará.

Finalmente, o tempo para obter a melhor solução aumentou significativamente para k=6, sendo que a justificação é a mesma para a diminuição do número de ciclos de relógio. Como o algoritmo demora mais tempo a ser executado por completo, o tempo de obtenção da melhor solução também aumenta.

Tarefa 2

Exercício 2a

Código:

Neste exercício os algoritmos são parecidos com os da alínea Exercício 1b, no entanto, em vez do objetivo ser minimizar a carga máxima das ligações, agora é minimizar a energia consumida pela rede. Para alcançar este objetivo, passamos a calcular dentro dos algoritmos não só as energias das ligações, mas também as dos routers.

No algoritmo "*Greedy Randomized*" recorremos à função 'calculateLinkLoads', já descrita na alínea Exercício 1c, para obter a energia das ligações. Sempre que esta energia tiver valor infinito, indica que excedeu a capacidade máxima por ligação de 100 Gbps, retornando também infinito para a energia total. Por outro lado, se for menor que infinito calculamos a energia gasta pelos routers recorrendo à função 'calculateNodesEnergy' para obter a energia total consumida, por forma a minimizá-la na condição seguinte.

Na otimização armazenamos as soluções que resultam na menor energia

consumida, ou seja, caso surja uma nova energia menor que a última guardada esta substitui os melhores resultados de consumo de energia obtidos.

Outra consideração a ter em conta nestes algoritmos que diferem do exercício anterior é que apenas guardamos a solução se os caminhos encontrados forem maiores que zero, caso contrário, retornamos infinito para a energia total significando que ainda não foi atribuído nenhum percurso ao fluxo.

```
function [sol,Loads,totalEnergy] = Task2a_GreedyRandomized_EnergyOptimized(nNodes,
Links,T,L,sP,nSP,sol)
    nFlows = size(T,1);
    sol = zeros(1,nFlows);

    for flow= randperm(nFlows) % rand order of flows
        path_index = 0;
        bestLoads = inf;
        best_energy = inf;

        for path = 1 : nSP(flow)
            sol(flow) = path;
            [Loads,linkEnergy] = calculateLinkLoads(nNodes, Links, T,L, sP, sol);
            % now we also need to take into account nodes energy
            if linkEnergy < inf
                nodesEnergy = calculateNodeEnergy(T,sP,nNodes,sol);
                totalEnergy = nodesEnergy + linkEnergy;
            else
                totalEnergy = inf;
            end

            if totalEnergy < best_energy
                path_index = path;
                bestLoads = Loads;
                best_energy = totalEnergy;
            end
        end

        if path_index > 0
            sol(flow) = path_index;
        else
            totalEnergy = inf;
            break;
        end
    end
    Loads = bestLoads;
    totalEnergy = best_energy;
end
```

Figura 12: GreedyRandomized_EnergyOptimized

No algoritmo "*Hill Climbing*" recebemos como argumento os resultados do "*Greedy Randomized*" dando sequência à otimização e adotámos a mesma lógica de fazer a soma de toda a energia consumida tanto dos routers como das ligações. Uma alteração importante efetuada neste algoritmo foi que quando se encontra um vizinho melhor e se atualizam os melhores resultados tem que se ter o cuidado das cargas não ultrapassarem os limites permitidos, se for o caso colocasse a carga com valor infinito e procura-se outra

válida, caso contrário guarda-se a maior carga.

```
function [sol,load,Loads,energy]= Task2a_hillClimbing_EnergyOptimized(sol,nNo-
des,Links,T,L,sP,nSP,energy,Loads)
    nFlows = size(T,1);
    %Loads= calculateLinkLoads(nNodes,Links,T,L,sP,sol);
    load= max(max(Loads(:,3:4)));

    LoadsBestNeigh = Loads;
    energyBestNeigh = energy;

    improved = true;
    while improved
        for flow= 1:nFlows
            for path= 1:nSP(flow) %check all neighbors
                if sol(flow)~=path % Não trocar a sol por ela mesma
                    auxsol = sol;
                    auxsol(flow)= path;
                    [auxLoads,auxLinkEnergy]= calculateLinkLoads(nNodes,Links,T,L,sP,auxsol);
                    nodesEnergy = calculateNodeEnergy(T,sP,nNodes,auxsol);
                    auxEnergy = nodesEnergy + auxLinkEnergy;

                    % check if the neighbour energy is better than the current one
                    if auxEnergy<energyBestNeigh
                        LoadsBestNeigh= auxLoads;
                        fbest = flow;
                        pbest = path;
                        energyBestNeigh= auxEnergy;
                    end
                end
            end
        end
        if energyBestNeigh<energy % Encontrado melhor vizinho, trocar valores
            sol(fbest)= pbest;
            Loads= LoadsBestNeigh;
            energy = energyBestNeigh;

            if Loads == inf
                load = inf;
            else
                load = max(max(Loads(:,3:4)));
            end
        else
            improved = false;
        end
    end
end
```

Figura 13: HillClimbing_EnergyOptimized

Exercício 2 b) e c)

Código:

Para estas alínea foram criados 2 scripts semelhantes à main da tarefa 1 para podermos aplicar os algoritmos referidos na alínea anterior, assim conseguimos comparar os resultados tanto para k=2 da alínea b) como para k=6 da alínea c).

Em baixo apresentamos essencialmente o excerto de código com alterações em relação à tarefa 1, salientando a comparação realizada a seguir à aplicação do algoritmo guardando os valores de energia apenas quando estes forem menores/melhores que os últimos guardados.

A primeira solução obtida pelo "Greedy Randomized" não pode ultrapassar a capacidade máxima de ligação de 100Gbps, logo corremos este algoritmo até obtermos uma solução viável.

Outra mudança, é que já não aparece o cálculo da energia dos nós no final do tempo limite, porque agora é feito dentro dos próprios algoritmos e já está associada à solução.

```
k=2
timeLimit = 60; % runtime limit of 60 seconds
bestEnergy = inf;
bestLoad = inf;
while toc(t) < timeLimit
    [sol, Loads, totalEnergy] = Task2a_GreedyRandomized_EnergyOptimized(nNodes, Links, T, L, sP,
nSP);
    % First solution must have maxLoad bellow max capacity (100)
    while totalEnergy == inf
        [sol, Loads, totalEnergy] = Task2a_GreedyRandomized_EnergyOptimized(nNodes, Links, T, L, sP,
nSP);
    end
    [sol, load, Loads, totalEnergy] = Task2a_hillClimbing_EnergyOptimized(sol, nNodes, Links, T, L,
sP, nSP, totalEnergy, Loads);
    if totalEnergy < bestEnergy
        bestSol = sol;
        bestLoad = load;
        bestLoads = Loads;
        bestEnergy = totalEnergy;
        bestTime = toc(t);
    end
    contador = contador + 1;
    somador=somador+load;
end
```

Figura 14: Alterações da task2b

Exercício 2d

Uma Nesta alínea reutilizou-se o código anterior alterando apenas o ‘k’ para 6.

Resultado:

```
For k= 2 and time=60
Worst link load of the solution = 99.80 Gbps
Average link load of the solution = 38.12 Gbps
Total network energy consumption of the solution = 2375.44
Average round-trip propagation delay of service 1 = 0.005384 sec
Average round-trip propagation delay of service 2 = 0.005751 sec
Number of links not supporting any traffic flow = 7
List of links not supporting any traffic flow:
    2 -> 3
    2 -> 5
    6 -> 8
    6 -> 15
    9 -> 10
    12 -> 13
    13 -> 15
Number of cycles run by the algorithm = 355
Time obtained best solution = 1.873409 sec
```

```
For k= 6 and time=60
Worst link load of the solution = 99.80 Gbps
Average link load of the solution = 37.21 Gbps
Total network energy consumption of the solution = 2096.67
Average round-trip propagation delay of service 1 = 0.005336 sec
Average round-trip propagation delay of service 2 = 0.006223 sec
Number of links not supporting any traffic flow = 11
List of links not supporting any traffic flow:
    1 -> 2
    1 -> 7
    2 -> 3
    3 -> 6
    3 -> 8
    4 -> 9
    6 -> 14
    6 -> 15
    12 -> 14
    13 -> 14
    14 -> 15
Number of cycles run by the algorithm = 253
Time obtained best solution = 48.497925 sec
```

Figura 15: Resultado do exercício 2c

Explicação:

Tal como explicamos nas conclusões da alínea 1 e) os ‘k’s indicam o número de caminhos candidatos à melhor solução e tanto com 2 como com 6 vemos que o resultado mantém-se constante quanto à carga máxima de ligação da solução obtida, o que é normal porque em ambos os casos é valorizada a minimização da energia e como para k=2 o valor da carga máxima já estava perto do máximo este manteve-se para k=6. O raciocínio é semelhante para a média das cargas das ligações que se manteve parecida para os 2 ‘k’s.

Com k=6 houve uma diminuição em termos de consumo de energia na rede isto deve-se à prioridade numa solução que minimize a energia consumida, logo quanto maior o leque de caminhos candidatos, menores os resultados obtidos como se pode verificar na imagem.

Ao nivel tempo médio dos atrasos de propagação de ida e volta para ambos os serviços sofreu um aumento, este fator acontece pelo simples motivo da variável ‘D’, que representa a propagação de atraso de cada ligação em cada direção dada pelo comprimento a dividir pela velocidade nas fibras ($v = 2\times10^5$ km/seg), ser proporcional ao comprimento das ligações, com k=6 temos mais caminhos (que serão maiores que k=2) disponíveis daí poderem ser escolhidas rotas maiores na solução levando a maiores atrasos.

Em relação ao número de ligações que não suportam qualquer tráfego estes valores aumentam quando temos mais caminhos candidatos, porque algumas ligações deixam de ser preferidas e entram em “modo de espera”. Com mais caminhos

candidatos a solução escolhe caminhos de menor consumo. Embora estes tenham menores gastos energéticos, esta solução pode levar ao aumento do congestionamento.

O número de ciclos preciso para percorrer os algoritmos e encontrar a melhor solução diminui para k=6, porque temos de analisar mais percursos todos nos algoritmos. Consequentemente demora mais a encontrar a solução ideal.

Exercício 2e

Neste exercício pediu-se para chamar os algoritmos anteriores, usando k = 6, ou seja, sendo previamente calculados os 6 caminhos candidatos para cada fluxo num tempo limite de 60 segundos.

Resultado:

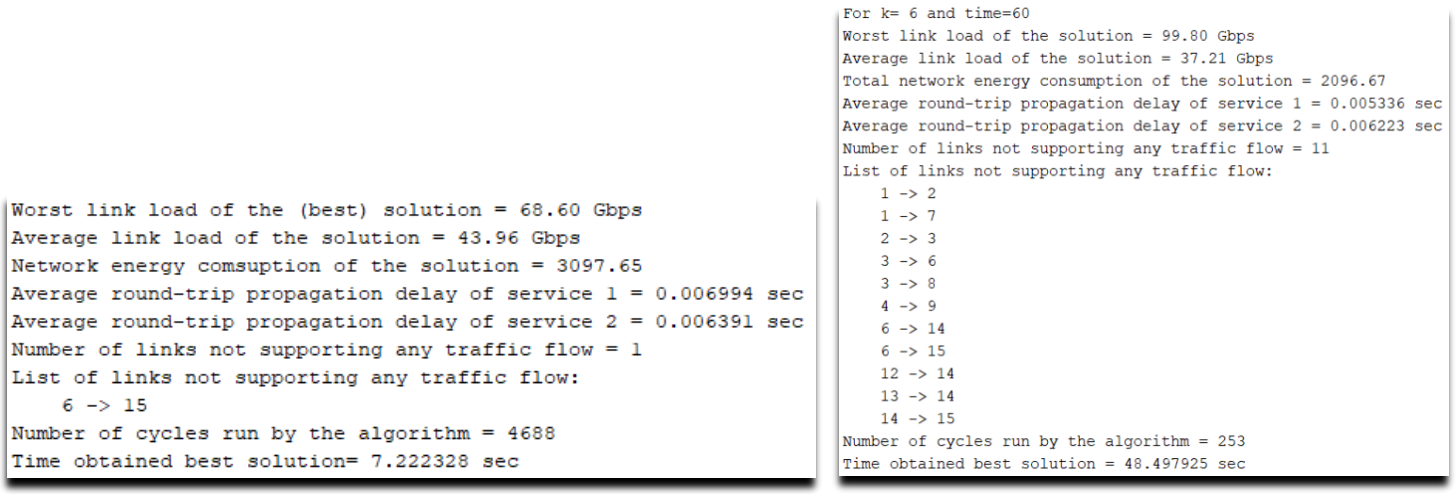

Figura 16: Resultado do exercício 2e

Explicação:

Ao testar o algoritmo especificado na alínea acima e ao compará-lo com o cenário sem a otimização energética, onde se recorreu ao exercício anterior, observou-se uma diminuição quanto ao consumo total de energia da rede, como o esperado.

A melhoria na eficiência energética da rede resultou, consequentemente, num aumento da carga máxima aproximando-se do limite suportado por cada ligação. O que faz sentido, porque com esta otimização priorizamos a eficiência da energia, independentemente das cargas.

Quanto à média da carga das ligações não há uma diferença significativa, o que é normal, porque apesar da carga máxima aumentar nada implica que a média varie.

Observando o tempo médio dos atrasos de propagação de ida e volta para ambos os serviços conseguimos ver à partida que com a otimização da energia este valor diminuiu ligeiramente, porque como explicamos na alínea Exercício 2d com a proporcionalidade do atraso com o comprimento das ligações, sabemos que vai

preferir ligações menores, consequentemente, levando a um menor atraso total dos serviços.

Existe um aumento do número de ligações que não suportam qualquer tráfego, isso pode ser devido ao algoritmo desativar mais ligações para economizar energia, colocando-os em “modo de espera”.

A diferença entre o número de ciclos a percorrer os algoritmos da tarefa 1 e 2, é consideravelmente menor porque ao agora acrescentamos o calculo da energia dos nós dentro do algoritmo, levando consequentemente ao aumento do tempo gasto para encontrar a melhor solução.

Tarefa 3

Exercício 3a

Código:

O algoritmo “*Greedy Randomized*” foi aproveitado da tarefa 1, já no “*Hill Climbing*” realizaram-se algumas alterações de acordo com o objetivo que era reduzir o pior atraso de ida e volta de um serviço e depois minimizar o atraso do outro serviço, em vez de ser minimizar a carga máxima ou a energia consumida da rede.

Para tal, a cada caminho por fluxo calculamos os atrasos correspondentes a cada serviço distinguindo o pior. Para este cálculo criámos uma função denominada por ‘*calculateServiceDelays*’ que aproveita o código da tarefa 1 usado na ‘main’ para calcular os atrasos no fim das otimizações como mostramos em baixo.

```
function [roundTripDelays] = calculateServiceDelays(sP, bestSol, D, T)
    nFlows = size(T,1);
    % Calculate round-trip propagation delay
    roundTripDelays = zeros(1,nFlows);

    for f = 1:nFlows
        if bestSol(f) > 0
            path= sP{f}{bestSol(f)}; % Para cada fluxo vamos buscar o caminho solucao
            for j=2:length(path)
                propagation_delay = D(path(j-1), path(j)); % D matrix represents propagation delays
                roundTripDelays(f) = roundTripDelays(f) + 2 * propagation_delay; % 2x because it's
round trip delay
            end
        end
    end
end
```

Figura 17: Cálculo dos atrasos

Depois de conhecer o pior serviço atribuímos as prioridades, obrigando que primeiro se faça a minimização do pior serviço e só depois é que se minimiza o outro serviço, para isso atribuímos uma prioridade de metade ao ‘outro’. Após estes cálculos começamos o processo de minimização de atrasos atualizando os melhores valores

comparando com os últimos guardados, se este for menor substitui o melhor e assim sucessivamente.

Em baixo segue o código relativo ao algoritmo modificado com as respetivas alterações.

```
function [sol, load,Loads,linkEnergy, delay]= Task3_HillClimbing_Otim_Delay(sol,
nNodes,Links,T,D,sP,nSP,L,nFlows1,linkEnergy)
    nFlows = size(T,1);
    T1_idx = 1:nFlows1;
    T2_idx = 1+nFlows1:nFlows;
    Loads= calculateLinkLoads(nNodes,Links,T,L,sP,sol);
    load= max(max(Loads(:,3:4)));
    bestDelay = calculateServiceDelays(sP, sol, D, T);
    delay = (sum(calculateServiceDelays(sP, sol, D, T)))/20;
    improved = true;
    while improved
        loadBestNeigh = inf;
        for flow= 1:nFlows
            for path= 1:nSP(flow) %Rodar por todos os vizinhos possiveis
                if sol(flow)~=path %Não trocar a sol por ela mesma
                    auxsol = sol;
                    auxsol(flow)= path;
                    [auxLoads,auxLinkEnergy]= calculateLinkLoads(nNodes,Links,T,L,sP,auxsol);
                    auxload= max(max(auxLoads(:,3:4)));

                    % delay of this sol
                    auxDelay = calculateServiceDelays(sP, auxsol, D, T);
                    if mean(auxDelay(T1_idx)) > mean(auxDelay(T2_idx))
                        auxDelayWorst = auxDelay(T1_idx);
                        auxDelayOther = auxDelay(T2_idx);
                    else
                        auxDelayOther = auxDelay(T1_idx);
                        auxDelayWorst = auxDelay(T2_idx);
                    end

                    % 0.5 * other, porque worst é mais prioritária
                    auxDelayLowPriority = 0.5 * sum(auxDelayOther);
                    auxDelayHighPriority = sum(auxDelayWorst);
                    valueAuxDelay = (auxDelayHighPriority + auxDelayLowPriority)/20;
                    valueBestDelay = (auxDelayHighPriority + auxDelayLowPriority)/20;

                    if valueAuxDelay < valueBestDelay
                        loadBestNeigh = auxload;
                        energyBestNeigh = auxLinkEnergy;
                        LoadsBestNeigh = auxLoads;
                        bestSol = auxsol;
                        bestDelay = auxDelay;
                    end
                end
            end
        end
        if max(bestDelay(T1_idx)) < delay % Encontrado melhor vizinho, trocar valores
            load = loadBestNeigh;
            Loads = LoadsBestNeigh;
            linkEnergy = energyBestNeigh;
            sol = bestSol;
            delay = max(bestNeighDelay(T1_idx));
        else
            improved = false;
        end
    end
end
```

Figura 18: Hill Climbing Task3

Exercício 3 b) e c)

Código:

Nesta alínea era apenas pedido para variar os caminhos candidatos para poder analisar nas alíneas seguintes k=2 da alínea b) como para k=6 da alínea c).

Em baixo apresentamos o script onde utilizámos o algoritmo “Multi Start”.

```
while toc(t) < timeLimit
    % sol = zeros(1,nFlows);

    [sol, ~, linkEnergy] = Task3_GreedyRandomized_Otim_Delay(nNodes,Links,T,L,sP,nSP);
    while linkEnergy == inf
        [sol, ~, linkEnergy] = Task3_GreedyRandomized_Otim_Delay(nNodes,Links,T,L,sP,nSP);
    end

    [sol, load, Loads, linkEnergy, delay] = Task3_HillClimbing_Otim_Delay(sol,nNo-
des,Links,T,D,sP,nSP,L,nFlows1,linkEnergy);

     if delay < bestDelay
        bestSol= sol;
        bestDelay = delay;
        bestLoad= load;
        bestLoads=Loads;
        bestLinkEnergy = linkEnergy;
        bestTime= toc(t);
    end
    contador= contador+1;
    somador= somador+load;
end

% Calculate average Link Load
link_load_sum = 0;
for i=1:nLinks
    link_load_sum = link_load_sum + sum(bestLoads(i,3:4));
end
avgLinkLoadSol = link_load_sum/nLinks;

% Calculate energy comsuption
nodesEnergy = calculateNodeEnergy(T,sP,nNodes,bestSol);
total_energy = nodesEnergy + bestLinkEnergy;

% Calculate round-trip propagation delay
T1_idx = 1:nFlows1;
T2_idx = 1+nFlows1:nFlows;
delays = calculateServiceDelays(sP, bestSol, D, T);
```

Figura 19: Alterações na task3

Exercício 3d

Os resultados apresentados nas figuras abaixo indicam as diferenças das métricas para k=2 e k=6.

Resultados:

```
For k= 2 and time=60
Worst link load of the (best) solution = 91.60 Gbps
Average link load of the solution = 36.70 Gbps
Network energy comsuption of the solution = 2747.07
Average round-trip propagation delay of service 1 = 0.005341 sec
Average round-trip propagation delay of service 2 = 0.005614 sec
Number of links not supporting any traffic flow = 5
List of links not supporting any traffic flow:
    1 -> 7
    3 -> 6
    3 -> 8
    6 -> 15
    12 -> 13
Number of cycles run by the algorithm = 3504
Time obtained best solution= 0.698917 sec
```

```
For k= 6 and time=60
Worst link load of the (best) solution = 82.40 Gbps
Average link load of the solution = 35.60 Gbps
Network energy comsuption of the solution = 2746.34
Average round-trip propagation delay of service 1 = 0.005077 sec
Average round-trip propagation delay of service 2 = 0.005687 sec
Number of links not supporting any traffic flow = 5
List of links not supporting any traffic flow:
    1 -> 7
    3 -> 6
    3 -> 8
    6 -> 15
    12 -> 13
Number of cycles run by the algorithm = 1628
Time obtained best solution= 1.205706 sec
```

Figura 20: Resultados Task3d

Explicação:

Dados mais caminhos curtos vemos desde já uma redução na carga máxima isto acontece porque há uma maior distribuição do tráfego e potencialmente reduz a carga na ligação máxima. O mesmo explica a média diminuir para k=6, uma maior distribuição do tráfego equilibra a carga das várias ligações.

O consumo de energia não é afetado pois esta otimização apenas valoriza os atrasos tanto para k=2 como para k=6 não interferindo no consumo energético na seleção dos caminhos.

Quando o 'k' é maior o algoritmo tem mais caminhos candidatos à disposição, desta forma pode reduzir os tempos de atraso, pois permite ao algoritmo ter mais flexibilidade ao contornar eventuais congestionamentos.

O número de ligações que não suportam qualquer tráfego permanece igual o que indica que estas ligações não são necessárias para a otimização dos atrasos independentemente da quantidade de caminhos candidatos.

Em relação ao tempo para obter a melhor solução, encontramos uma tempo maior para k=6 o que é normal pois tem mais percursos para analisar e escolher o que prefere, o que implica que percorre menos ciclos passando mais tempo em cada um.

Exercício 3e

Relacionando os resultados das tarefas anteriores com esta conseguimos concluímos que a propagação do atraso médio de ida e volta de cada serviço é menor que ambas as tarefas anteriores, tal como esperado.

Tabela 1: Resultados Task3e

Resultados	1d	2c	3c
Worst link load (Gbps)	68,60	99,80	82,40
Average link load (Gbps)	43,96	37,21	35,60
Network energy comsuption	3097,65	2096,67	2746,34
Average round-trip propagation delay of service 1 (ms)	6,99	5,34	5,08
Average round-trip propagation delay of service 2 (ms)	6,39	6,22	5,69
Number of links not supporting any traffic flow	1,00	11,00	5,00
Number of cycles run by the algorithm	4688,00	253,00	1628,00
Time to obtained best solution (s)	7,22	48,49	1,20

Quanto à pior carga, já não sobrecarrega a capacidade das ligações como na otimização da energia, contudo continua pior do que a otimização da carga máxima como é previsto, pois não é o principal foco. O mesmo acontece com a média das cargas, esta diminui em relação aos outros resultados até mesmo relação à otimização da carga máxima, pois esta valoriza a pior carga em vez da média.

Como explicámos para a otimização da energia o atraso de propagação de ida e volta dos serviços é proporcional ao comprimento das ligações, portanto, esta como tem ligações menores consome menos energia em relação à primeira tarefa, contudo, como era de esperar permaneceu maior que o consumo de energia da tarefa que prioriza a energia.

Quanto ao número de ligações que não suportam qualquer tráfego obtemos um valor intermédio em relação às outras tarefas, pois os atrasos não variam tanto como as energias requeridas em cada ligação, mas ainda assim, existem algumas ligações que prejudicam a eficiência da rede, passando a estar em '*Sleeping Mode*'.

No que diz respeito ao número de ciclos a correr o algoritmo, estes de igual forma que a otimização de energia tem menos ciclos, pois dentro do algoritmo temos que realizar os cálculos dos atrasos para cada serviço, em relação à tarefa 1, mas não tanto como na tarefa 2, porque não consideramos os router, por sua vez, como não consideramos a possibilidade de exceder a capacidade de ligação onde ficávamos a correr enquanto a energia não fosse viável, a otimização do atraso não perdia tempo neste processo.

Tarefa 4

Exercício 4a

Código:

Com o objetivo de adaptar o algoritmo para também suportar o serviço ‘anycast’ realizámos alterações no código principal e criámos uma função “bestCostPaths”.

O código da função está presente abaixo:

```
function [sP, nSP] = bestCostPaths(nNodes, anycastNodes, L, T_anycast)
    % k=1, only shortest path
    % Best Cost Paths for each node. Determine which anycast node
    % should be the destination for each source node
    k= 1;
    sP= cell(1, nNodes);
    nSP= zeros(1,nNodes);
    for n = 1:nNodes
        if ismember(n, anycastNodes)
            if ismember(n, T_anycast(:,1))
            % if node is in anycast matrix and is
            % anycast node, sP has source and dest
            % as itself
                sP{n} = {[n n]};
                nSP(n) = 1;
            else
                nSP(n) = -1;
            end
            continue;
        end

        if ~ismember(n, T_anycast(:,1)) % node needs to be in T3(the only anycast service)
            nSP(n) = -1;
            continue;
        end

        best = inf;
        for a = 1:length(anycastNodes)
            % Get SP and Cost for each destination (any cast) node
            [shortestPath, totalCost] = kShortestPath(L, n, anycastNodes(a), k);

            if totalCost < best
                sP{n}= shortestPath;
                nSP(n)= length(totalCost);
                best = totalCost;
            end
        end
    end
    nSP= nSP(nSP~=-1);
    sP= sP(~cellfun(@isempty, sP));
end
```

Figura 21: Obter caminhos com melhor custo

O propósito desta função é encontrar o nó de destino, ‘anycast’, para cada um dos nós não ‘anycast’. Os nós são percorridos um a um, sendo que a primeira coisa verificada é se o nó é ‘anycast’ ou não. Se for e se pertencer à matriz ‘anycast’(T3) como nó de origem, o caminho mais curto para o nó, começa e acaba nele mesmo. Se

o nó for ‘anycast’, mas não pertencer à matriz como nó de origem não é calculado caminho mais curto para ele.

Para os nós que são origem e não são destino são calculadas as possibilidades de caminhos mais curtos para cada um dos nós ‘anycast’, por forma a que o caminho guardado e o nó de destino escolhido sejam os que proporcionam menor custo.

Quanto ao código principal foi necessário separar o cálculo dos caminhos, primeiro para os serviços ‘unicast’ e depois para o serviço ‘anycast’:

```
    % Computing up to k=6 shortest paths for all flows:
    k= 6;
    sP_Unicast= cell(1,nFlowsUnicast);
    nSP_Unicast= zeros(1,nFlowsUnicast);
    for f=1:nFlowsUnicast
        [shortestPath, totalCost] = kShortestPath(L,T_Unicast(f,1),T_Unicast(f,2),k);
        sP_Unicast{f}= shortestPath;
        nSP_Unicast(f)= length(totalCost);
    end

    [sP_Anycast,nSP_Anycast]= bestCostPaths(nNodes,anycastNodes,L,T3);

    % Add destination nodes no anycast matrix T3
    T3 = [T3(:,1) zeros(size(T3,1),1) T3(:,2:3)];
    for i= 1:nFlowsAnycast
        T3(i,2) = sP_Anycast{i}{1}(end);
    end
```

Figura 22: Obter caminhos, concatenações

Depois de obtidos os caminhos únicos para o serviço ‘anycast’ é necessário adicionar os nós de destino encontrados para cada nó origem da matriz T3.

Já com a matriz T3 completa, podemos concatenar esta com a matriz ‘unicast’. Os caminhos calculados para todos os serviços também são concatenados. Finalmente é possível correr o algoritmo para encontrar a solução que minimize a energia consumida pela rede.

```
% Concatenate unicast and anycast T's, sP's and nSP's
T = [T_Unicast; T3];
sP = cat(2, sP_Unicast, sP_Anycast);
nSP = cat(2, nSP_Unicast, nSP_Anycast);

t = tic;
timeLimit = 60; % runtime limit of 60 seconds
bestEnergy = inf;
contador = 0;
while toc(t) < timeLimit
    [sol, Loads, totalEnergy] = Task4_GreedyRandomized_EnergyOptimized(nNodes, Links, T,
L, sP, nSP);

    while totalEnergy == inf
        [sol, Loads, totalEnergy] = Task4_GreedyRandomized_EnergyOptimized(nNodes, Links,
T, L, sP, nSP);
    end

    [sol, load, Loads, totalEnergy] = Task4_hillClimbing_EnergyOptimized(sol, nNodes,
Links, T, L, sP, nSP, totalEnergy, Loads);

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

Figura 23: Alterações Task4

Exercício 4b

Código já realizado no 4 a).

Exercício 4c

Código

Para encontrar a combinação de nós anycast que minimiza a médio do atraso de propagação de ida e volta implementámos o seguinte código:

```
anycastPairs = nchoosek(1:15,2);

bestRoundTripDelay = inf;
bestPair = [0 0];
for pair = 1:size(anycastPairs,1)
    anycastNodes = anycastPairs(pair,:);

    [sP_Anycast,nSP_Anycast]= bestCostPaths(nNodes,anycastNodes,L,T3);

    % Pair that minimizes average round trip delay of the anycast service
    roundTripDelays = zeros(1,nFlowsAnycast);
    for f = 1:nFlowsAnycast
        path= sP_Anycast{f}{1};

        total_delay = 0;
        for j=2:length(path)
            propagation_delay = D(path(j-1), path(j)); % D matrix represents propagation
delays
            total_delay = total_delay + propagation_delay;
        end
        roundTripDelays(f) = 2*total_delay; % 2x because it's round trip delay
    end
    totalRoundTripDelay = sum(roundTripDelays);
    averageRoundTripDelay = totalRoundTripDelay / nFlowsAnycast;

    if averageRoundTripDelay < bestRoundTripDelay
        bestRoundTripDelay = averageRoundTripDelay;
        bestPair = anycastNodes;
    end
end

fprintf("Melhor par: %d,%d\n",bestPair(1),bestPair(2));
```

Figura 24: Obter melhor par 'anycast'

A variável anycastPairs contém todas as combinações possíveis para nós anycast. Para cada uma das combinações são calculados os caminhos mais curtos para os nós destino e, com estes, obtemos o atraso de propagação para o serviço ‘anycast’. Se o atraso for menor que o melhor anteriormente guardado, o novo par é guardado como o melhor.

Chegámos à conclusão de que o melhor par era os nós 7 e 8.

Exercício 4d

Foi corrido o exercício 4 b), mas desta vez com o par de nós 7 e 8 como anycast nodes.

Exercício 4e

Resultados

```
Worst link load of the solution = 99.80 Gbps
Average link load of the solution = 40.29 Gbps
Total network energy consumption of the solution = 2445.16
Average round-trip propagation delay of unicast service 1 = 5.444167 sec
Average round-trip propagation delay of unicast service 2 = 6.380000 sec
Average round-trip propagation delay of anycast service 3 = 3.547778 sec
Number of links not supporting any traffic flow = 7
List of links not supporting any traffic flow:
    1 -> 2
    2 -> 3
    3 -> 8
    12 -> 13
    12 -> 14
    13 -> 14
    14 -> 15
Number of cycles run by the algorithm = 563
Time obtained best solution = 52.073033 sec
```

Figura 25: Valores para 4 b), anycast nodes = [3 10]

```
Worst link load of the solution = 99.80 Gbps
Average link load of the solution = 41.99 Gbps
Total network energy consumption of the solution = 2262.89
Average round-trip propagation delay of unicast service 1 = 5.718333 sec
Average round-trip propagation delay of unicast service 2 = 6.243750 sec
Average round-trip propagation delay of anycast service 3 = 2.538889 sec
Number of links not supporting any traffic flow = 9
List of links not supporting any traffic flow:
    1 -> 5
    2 -> 3
    2 -> 5
    3 -> 6
    3 -> 8
    6 -> 14
    6 -> 15
    13 -> 14
    13 -> 15
Number of cycles run by the algorithm = 501
Time obtained best solution = 30.083445 sec
```

Figura 26: Valores para 4 d), anycast nodes = [7 8]

Conclusões

Ambos os valores de pior carga máxima e carga média das ligações permanecerem praticamente iguais devido ao facto de ambas as soluções valorizarem a minimização do consumo de energia, levando a que em ambos os casos a carga máxima já esteja muito próxima do limite.

O valor da energia desceu um pouco para o par 7 e 8 de nós. Como é possível ver temos menos ligações ativas, o que leva a uma diminuição da energia.

Quanto ao atraso médio de propagação de ida e volta dos serviços, vemos um aumento relativamente maior no serviço unicast 1, para o segundo par de nós. Este segundo par tinha como foco diminuir o atraso de propagação apenas para o serviço anycast, 3, pelo que já era expectável algum dos atrasos dos outros serviços subir.

Para diminuir o atraso de propagação, com valores presentes na matriz D, é necessário procurar caminhos menores. Consequentemente, obtemos mais ligações inativas.

A performance do algoritmo quanto aos ciclos permanece muito semelhante, já os tempos de obtenção da melhor solução variam. Isto deve-se inalteração no código dos algoritmos, que executam a mesma tarefa para ambos os casos (o valor de 'k' permanece igual).

Contribuição dos autores

Tiago Alves – 50 %
Rafael Amorim – 50 %