Universidade de Aveiro

Modelação e Desempenho de Redes e Seviços

Mini-Project nr.2

André Clérigo (98485), Pedro Rocha (98256)

Departamento de Eletrónica, Telecomunicações e Informática

January 9th, 2023

Contents

Chapter 1

Task 1

1.1 Exercise 1.a)

1.1.1 Code

We start by setting variables for the number of nodes, number of flows (unicast and anycast), destination nodes for anycast service, link and node capacity.

After that we will calculate the shortest paths (sP) for each type of service, for the unicast service we use the algorithm practiced in the practical classes, for the anycast service we did a custom function *bestAnycastPaths.m* which will be explained in this report afterwards.

Meanwhile, we need to add a new 2nd column to the T_any matrix that indicates the best destination node (based on sP_any calculated previously), after this, we are ready to concatenate T_uni and T_any to create a matrix (T) with all flows (unicast and anycast service).

Finally, we are ready to calculate the link loads, link energy, and node energy of our system.

```
% Initial variables
load('InputDataProject2.mat');
nNodes = size(Nodes,1);
nFlows_uni = size(T_uni, 1);
nFlows_any = size(T_any, 1);
lc = 50;          % Link capacity of 50Gbps
nc = 500;         % Node capacity of 500Gbps
anycastNodes = [5 12];

% Traffic flows for unicast service
% Computing up to k=1 shortest path for all flows
k = 1;
sP_uni = cell(1, nFlows_uni);
nSP_uni = zeros(1, nFlows_uni);
% sP{f}{i} is the i-th path of flow f
```

```
% nPS{f}{i} is the number of paths of flow f

for f = 1 : nFlows_uni
    [shortestPath, totalCost] = kShortestPath(L, ...
        T_uni(f,1), T_uni(f,2), k);
    sP_uni{f} = shortestPath;
    nSP_uni(f)= length(totalCost);
end
% Traffic flows for anycast service
[sP_any, nSP_any] = bestAnycastPaths(nNodes, anycastNodes, ...
    L, T_any);

% Reconstructing T matrix
% srcNode dstNode upRate dwRate
T_any = [T_any(:, 1) zeros(size(T_any,1), 1) T_any(:, 2:3)];
for i = 1 : size(T_any, 1)
    T_any(i, 2) = sP_any{i}{1}(end);
end

% Calculate general T, sP and nSP
T = [T_uni; T_any];
sP = cat(2, sP_uni, sP_any);
nSP = cat(2, nSP_uni, nSP_any);

sol = ones(1, nFlows_uni + nFlows_any);
[Loads, linkEnergy] = calculateLinkLoadEnergy(nNodes, ...
    Links, T, sP, sol, L, lc);
maxLoad = max(max(Loads(:,3:4)));

for i = 1 : length(Loads)
    fprintf('{%d - %d}:\t%.2f\t%.2f\n', Loads(i), Loads(i, ...
        2), Loads(i, 3), Loads(i, 4))
end
fprintf('Worst Link Load: %.2f Gbps\n', maxLoad);
```

For the anycast service, the function iterates through all possible nodes and, if this node is a destination node or does not belong to an anycast flow, we ignore it, for the rest we get the shortest path twice, once for each possible destination node and keep the best value.

```
function [sP, nSP] = bestAnycastPaths(nNodes, anycastNodes, ...
    L, T_any)
    sP = cell(1, nNodes);
    nSP = zeros(1, nNodes);
    for n = 1:nNodes
        if ismember(n, anycastNodes)    % if the node is a ...
            anycastNode skip it
            nSP(n) = -1;
            continue;
        end

        if ¬ismember(n, T_any(:, 1))     % node is not from ...
```

```
                T_any matrix
                nSP(n) = -1;
                continue;
            end

            best = inf;
            for a = 1:length(anycastNodes)
                [shortestPath, totalCost] = kShortestPath(L, n, ...
                    anycastNodes(a), 1);

                if totalCost(1) < best
                    sP{n} = shortestPath;
                    nSP(n) = length(totalCost);
                    best = totalCost;
                end
            end
        end

        nSP = nSP(nSP≠-1);                  % remove unwanted values
        sP = sP(¬cellfun(@isempty, sP));     % remove empty ...
            entry from the cell array
    end
```

Our *calculateLinkLoadEnergy.m* is a modification of *calculateLinkLoad.m* used in the practical classes, with the difference being that, besides calculating the Loads, we also calculate the energy of the links with the given expressions.

$$E_l = \begin{cases} 2, \text{if link in sleeping mode} \\ 6 + 0.2 * l, \text{if link is active} \end{cases}$$

```
function [Loads, linkEnergy] = ...
    calculateLinkLoadEnergy(nNodes, Links, T, sP, Solution, ...
    L, capacity)
    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)];
    linkEnergy = 0;
```

```
    for i= 1:nLinks
        Loads(i,3)= aux(Loads(i,1),Loads(i,2));
        Loads(i,4)= aux(Loads(i,2),Loads(i,1));
        maxLoad = max(max(Loads(:, 3:4)));
        % If the worst link load is greater than max ...
            capacity , energy will be infinite
        if maxLoad > capacity
            linkEnergy = inf;
        else
            % link in sleeping mode
            if max(Loads(i, 3:4)) == 0
                linkEnergy = linkEnergy + 2;         % El = ...
                    2 whatever the link capacity
            else
                % len from nodeA to nodeB
                len = L(Loads(i, 1), Loads(i, 2));
                % energy calculation dependent of link capacity
                if capacity == 50
                    linkEnergy = linkEnergy + 6 + 0.2 * len;
                elseif capacity == 100
                    linkEnergy = linkEnergy + 8 + 0.3 * len;
                else
                    fprintf('Error: Link capacity is not ...
                        50Gbps nor 100Gbps\n');
                end
            end
        end
    end
end
```

Our *calculateNodeEnergy* function creates a matrix of one line with all the traffic supported by each node, which is done by iterating through all flows and, for each flow, sum its throughput (in both directions) at each node that belongs to, to the flow's forwarding path (sP), then we calculate the energy with the given expression.

$$E_n = 10 + 90t^2$$

```
function energy = calculateNodeEnergy(T, sP, nNodes, nc, sol)
    nodesTraffic = zeros(1, nNodes);
    for flow = 1 : size(T,1)
        if sol(flow) ≠ 0
            nodes  = sP{flow}{sol(flow)};
            for n = nodes
                nodesTraffic(n) = nodesTraffic(n) + ...
                    sum(T(flow, 3:4));
            end
        end
    end
    energy = sum(10 + 90 * (nodesTraffic/nc).^2);
end
```

1.1.2 Results

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

Worst Link Load: 49.90 Gbps

1.2 Exercise 1.b)

1.2.1 Code

The following snippet of code needs to be executed after ex1.a) to have access to the correct value of *Loads*, *nNodes*, *nc* (node capacity) and *linkEnergy*.

```
sleepingLinks = '';
for i = 1 : size(Loads, 1)
    if max(Loads(i, 3:4)) == 0
        sleepingLinks = append(sleepingLinks, ' {', ...
            num2str(Loads(i,1)), ', ', num2str(Loads(i,2)), ...
            '}');
    end
end

nodeEnergy = calculateNodeEnergy(T, sP, nNodes, nc, sol);

fprintf('Network energy consumption: %.2f\n', linkEnergy + ...
    nodeEnergy);
fprintf('List of links in sleeping mode:%s\n', sleepingLinks);
```

1.2.2 Results

Network energy consumption: 851.81
List of links in sleeping mode: {6, 8}

1.3 Exercise 1.c)

1.3.1 Code

To develop a Multi Start Hill Climbing algorithm with initial Greedy Randomized solutions, we started by creating a *greedyRandomizedStrategy.m*, inspired in the practical classes.

The function first generates a random permutation of the flows and then iterates through each flow. For each flow, it tries each of the available paths and calculates the maximum link load and the total energy consumption of the links for that path. It chooses the path that results in the lowest maximum link load, and updates the solution with the index of that path. The function then repeats this process for the next flow in the random permutation.

Finally, the function returns the resulting routing solution, the link loads, maximum link load and energy consumption of the links.

```
function [sol, Loads, maxLoad, linkEnergy] = ...
    greedyRandomizedStrategy(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_maxLoad = 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
            [Loads, linkEnergy] = ...
                calculateLinkLoadEnergy(nNodes, Links, T, ...
                sP, sol, L, 50);
            maxLoad = max(max(Loads(:, 3:4)));

            % check if the current load is better then bestLoad
            if maxLoad < best_maxLoad
                % change index of path and load
                path_index = path;
                best_maxLoad = maxLoad;
                best_Loads = Loads;
                best_energy = linkEnergy;
            end
        end
```

```
            sol(flow) = path_index;
        end
        Loads = best_Loads;
        maxLoad = max(max(Loads(:, 3:4)));
        linkEnergy = best_energy;
    end
```

Following the same principle for *HillClimbingStrategy.m*, we used the developed function in the practical classes, changing it to calculate the energy consumption.

The function tries to find a routing solution that minimizes the maximum link load, by repeatedly trying to improve the current routing solution by iterating through each flow and each available path for that flow, and calculating the maximum link load for that path. If it finds a path that results in a lower maximum link load than the current solution, it updates the routing solution, link loads, and energy consumption with the new values. The function continues this process until it can no longer find a path that improves the solution.

Finally, the function returns the resulting routing solution, the link loads, maximum link load and energy consumption.

```
function [sol, Loads, maxLoad, linkEnergy] = ...
    HillClimbingStrategy(nNodes, Links, T, sP, nSP, sol, ...
    Loads, linkEnergy, L)
    nFlows = size(T,1);
    % set the best local variables
    maxLoad = max(max(Loads(:, 3:4)));
    bestLocalLoad = maxLoad;
    bestLocalLoads = Loads;
    bestLocalSol = sol;
    bestLocalEnergy = linkEnergy;

    % 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
                    [auxLoads, auxLinkEnergy] = ...
                        calculateLinkLoadEnergy(nNodes, ...
                        Links, T, sP, auxSol, L, 50);
                    auxMaxLoad = max(max(auxLoads(:, 3:4)));

                    % check if the current load is better ...
```

```
                            then start load
                        if auxMaxLoad < bestLocalLoad
                            bestLocalLoad = auxMaxLoad;
                            bestLocalLoads = auxLoads;
                            bestLocalSol = auxSol;
                            bestLocalEnergy = auxLinkEnergy;
                        end
                    end
                end
            end

            if bestLocalLoad < maxLoad
                maxLoad = bestLocalLoad;
                Loads = bestLocalLoads;
                sol = bestLocalSol;
                linkEnergy = bestLocalEnergy;
            else
                improved = false;
            end
        end
    end
```

1.4 Exercise 1.d)

1.4.1 Code

The first 41 lines of the following code snippet are the same as it was shown in exercise 1.a) (except the value of k, which is 2, now).

The greedy randomized strategy is used to generate an initial solution, and then the hill climbing strategy is used to try to improve the solution. The process is repeated multiple times within a time limit, and the best solution found is returned.

The initial solution is generated by calling the *greedyRandomizedStrategy* function, which returns a routing solution (sol), the link loads (*Loads*), the maximum link load (*maxLoad*), and the total energy consumption of the links (*linkEnergy*). The hill climbing strategy is then applied to the initial solution by calling the *HillClimbingStrategy* function, which returns an improved routing solution *sol* (or not if the greedy one was the best solution, which is highly unlikely), the link loads (*Loads*), the maximum link load (*maxLoad*), and the energy consumption of the links (*linkEnergy*).

The resulting solution is compared to the current best solution, and if it is better, it is stored as the new best solution. The process is repeated until the time limit is reached.

Finally, the energy consumption of the nodes is calculated by calling the *calculateNodeEnergy* function, and the total energy consumption of the network is calculated as the sum of the energy consumption of the links and the nodes.

```
1  % Initial variables
2  load('InputDataProject2.mat');
3  nNodes = size(Nodes,1);
4  nFlows_uni = size(T_uni, 1);
5  lc = 50;         % Link capacity of 50Gbps
6  nc = 500;        % Node capacity of 500Gbps
7  anycastNodes = [5 12];
8
9  % Traffic flows for unicast service
10 % Computing up to k=1 shortest path for all flows
11 k = 2;
12 sP_uni = cell(1, nFlows_uni);           % sP{f}{i} is the ...
       i-th path of flow f
13 nSP_uni = zeros(1, nFlows_uni);         % nPS{f}{i} is the ...
       number of paths of flow f
14 for f = 1 : nFlows_uni
15     [shortestPath, totalCost] = kShortestPath(L, ...
           T_uni(f,1), T_uni(f,2), k);
16     sP_uni{f} = shortestPath;
17     nSP_uni(f)= length(totalCost);
18 end
19 % Traffic flows for anycast service
```

```
[sP_any, nSP_any] = bestAnycastPaths(nNodes, anycastNodes, ...
    L, T_any);
% Reconstructing T matrix
% srcNode dstNode upRate dwRate
T_any = [T_any(:, 1) zeros(size(T_any,1), 1) T_any(:, 2:3)];
for i = 1 : size(T_any, 1)
    T_any(i, 2) = sP_any{i}{1}(end);
end
% Calculate general T, sP and nSP
T = [T_uni; T_any];
sP = cat(2, sP_uni, sP_any);
nSP = cat(2, nSP_uni, nSP_any);

t = tic;
timeLimit = 30;
bestLoad = inf;
bestLinkEnergy = inf;
contador = 0;
while toc(t) < timeLimit
    % greedy randomzied start
    [sol, Loads, maxLoad, linkEnergy] = ...
        greedyRandomizedStrategy(nNodes, Links, T, sP, nSP, L);
    % The first solution should have a maxLinkLoad bellow ...
        the maxmium link
    % capacity
    while maxLoad > lc
        [sol, Loads, maxLoad, linkEnergy] = ...
            greedyRandomizedStrategy(nNodes, Links, T, sP, ...
            nSP, L);
    end
    [sol, Loads, maxLoad, linkEnergy] = ...
        HillClimbingStrategy(nNodes, Links, T, sP, nSP, ...
        sol, Loads, linkEnergy, L);
    if maxLoad < bestLoad
        bestSol = sol;
        bestLoad = maxLoad;
        bestLoads = Loads;
        bestLinkEnergy = linkEnergy;
        bestLoadTime = toc(t);
    end
    contador = contador + 1;
end
nodeEnergy = calculateNodeEnergy(T, sP, nNodes, nc, bestSol);
energy = bestLinkEnergy + nodeEnergy;

sleepingLinks = '';
for i = 1 : size(Loads, 1)
    if max(Loads(i, 3:4)) == 0
        sleepingLinks = append(sleepingLinks, ' {', ...
            num2str(Loads(i,1)), ', ', num2str(Loads(i,2)), ...
            '}');
    end
end

```

```
fprintf("E = %.2f \t W = %.2f \t No. sols = %d \t time = ...
    %.2f\n", energy, bestLoad, contador, bestLoadTime);
fprintf('List of links in sleeping mode:%s\n', sleepingLinks);
```

1.4.2 Results and Conclusions

Energy	Worst Link Load (Gbps)	No. Solutions	Time (s)
896.09	40.60	5322	0.03

List of links in sleeping mode: Empty

Comparing our results with exercises 1.a) and 1.b) we see that we achieved a better worst link load of 40.6 Gbps when compared with a worst link load of 49.90 Gbps on exercise 1.a), meanwhile the network energy consumption of this algorithm was 896.09, which is higher than the one we got on exercise 1.b), that was 851.81.

In 1.a), the shortest path algorithm is used to find the path with the lowest cost between the source and destination of each traffic flow. This can lead to a solution that has a higher maximum link load because the shortest path may not always be the path with the lowest load on the links. For example, if there is a link that is heavily loaded, the shortest path may go through it, leading to a higher load on that link. In 1.d), the value of k is equal to 2 in the shortest path calculations, while 1.a) uses k equal to 1. With a larger value of k, the algorithm has more paths to choose from, which can potentially lead to a better solution.

The hill climbing algorithm in 1.d) starts with a random initial solution and it iteratively makes small changes in an attempt to improve it. If the change results in a lower maximum link load, it is accepted, however, if it results in a higher maximum link load, it is rejected. This can lead to a solution with a lower maximum link load because the algorithm is able to explore different paths for the flows and select the ones that result in the lowest load on the links.

On the other hand, the link energy in 1.d) is higher than in 1.b) because the hill climbing algorithm is not specifically developed to minimize energy consumption. It is only focused on optimizing the maximum link load, and as a result, it may select paths that have a higher energy consumption but a lower maximum link load.

Another thing to note is that the value of the network energy consumption isn't always the same, this is because there can be similar solutions with the same worst link load that generate different network energy consumption.

1.5 Exercise 1.e)

1.5.1 Code

The following code snippet is the same as it was shown in exercise 1.d) (except the value of k, which is 6, now).

```
% Initial variables
load('InputDataProject2.mat');
nNodes = size(Nodes,1);
nFlows_uni = size(T_uni, 1);
lc = 50;          % Link capacity of 50Gbps
nc = 500;         % Node capacity of 500Gbps
anycastNodes = [5 12];

% Traffic flows for unicast service
% Computing up to k=1 shortest path for all flows
k = 6;
sP_uni = cell(1, nFlows_uni);            % sP{f}{i} is the ...
    i-th path of flow f
nSP_uni = zeros(1, nFlows_uni);          % nPS{f}{i} is the ...
    number of paths of flow f
for f = 1 : nFlows_uni
    [shortestPath, totalCost] = kShortestPath(L, ...
        T_uni(f,1), T_uni(f,2), k);
    sP_uni{f} = shortestPath;
    nSP_uni(f)= length(totalCost);
end
% Traffic flows for anycast service
[sP_any, nSP_any] = bestAnycastPaths(nNodes, anycastNodes, ...
    L, T_any);


% Reconstructing T matrix
% srcNode dstNode upRate dwRate
T_any = [T_any(:, 1) zeros(size(T_any,1), 1) T_any(:, 2:3)];
for i = 1 : size(T_any, 1)
    T_any(i, 2) = sP_any{i}{1}(end);
end
% Calculate general T, sP and nSP
T = [T_uni; T_any];
sP = cat(2, sP_uni, sP_any);
nSP = cat(2, nSP_uni, nSP_any);

t = tic;
timeLimit = 30;
bestLoad = inf;
bestLinkEnergy = inf;
contador = 0;
while toc(t) < timeLimit
    % greedy randomzied start
    [sol, Loads, maxLoad, startLinkEnergy] = ...
```

```
        greedyRandomizedStrategy(nNodes, Links, T, sP, nSP, L);
    % The first solution should have a maxLinkLoad bellow ...
        the maxmium link
    % capacity
    while maxLoad > lc
        [sol, Loads, maxLoad, startLinkEnergy] = ...
            greedyRandomizedStrategy(nNodes, Links, T, sP, ...
            nSP, L);
    end
    [sol, Loads, maxLoad, linkEnergy] = ...
        HillClimbingStrategy(nNodes, Links, T, sP, nSP, ...
        sol, Loads, startLinkEnergy, L);

    if maxLoad < bestLoad
        bestSol = sol;
        bestLoad = maxLoad;
        bestLoads = Loads;
        bestLinkEnergy = linkEnergy;
        bestLoadTime = toc(t);
    end
    contador = contador + 1;
end

nodeEnergy = calculateNodeEnergy(T, sP, nNodes, nc, bestSol);
energy = bestLinkEnergy + nodeEnergy;

sleepingLinks = '';
for i = 1 : size(Loads, 1)
    if max(Loads(i, 3:4)) == 0
        sleepingLinks = append(sleepingLinks, ' {', ...
            num2str(Loads(i,1)), ', ', num2str(Loads(i,2)), ...
            '}');
    end
end

fprintf("E = %.2f \t W = %.2f \t No. sols = %d \t time = ...
    %.2f\n", energy, bestLoad, contador, bestLoadTime);
fprintf('List of links in sleeping mode:%s\n', sleepingLinks);
```

1.5.2 Results and Conclusions

Energy	Worst Link Load (Gbps)	No. Solutions	Time (s)
896.70	40.60	1948	0.03

List of links in sleeping mode: {6, 8}

Increasing the value of k from 2 to 6 in the shortest path calculations has resulted in the same maximum link load, and has not affected the link energy.

There is, also, a variation on the list of sleeping links and it may be due to the randomized nature of the algorithm. With this in mind, and since this algorithm isn't optimized to get the best energy consumption, there may be links that don't affect the maximum link load and, therefore the load on that link isn't considered.

Generally, increasing the value of k allows the algorithm to explore more paths of each flow, which can potentially lead to better solutions in terms of maximum link load. In this particular case, it is possible that the paths with the lowest maximum link load are always the shortest or the second shortest path, or that using other paths results in the same maximum link load.

It is to note that, even though the best maximum link load for this configurations is always the same, the number of solutions and the solutions themselves are different. Therefore, similarly to the previous exercise, the network energy consumption fluctuates.

Chapter 2

Task 2

2.1 Exercise 2.a)

2.1.1 Code

In order to optimize the problem by minimizing the energy consumption of the network, we had to modify *greedyRandomizedStrategy.m* and *HillClimbingStrategy.m*.

For the *greedyRandomizedStrategy.m* we completely disregard the variables *best_maxLoad* and *maxLoad*, because we are trying to minimize the energy; however, we also have to check if the maximum load is infinite before returning the values. We also check if the final path_index is 0, if it is than there is no optimal solution, we set the energy to infinite and stop the process.

```
function [sol, Loads, energy] = ...
    greedyRandomizedStrategy(nNodes, Links, T, sP, nSP, L)
    Loads = inf;
    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_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
            [Loads, linkEnergy] = ...
```

```
                    calculateLinkLoadEnergy(nNodes, Links, T, ...
                    sP, sol, L, 50);
                if linkEnergy < inf
                    nodeEnergy = calculateNodeEnergy(T, sP, ...
                        nNodes, 500, sol);
                    energy = linkEnergy + nodeEnergy;
                else
                    energy = inf;
                end

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

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

Taking a similar approach for *HillClimbingStrategy.m*, we removed the variables *BestLocalLoad* and *auxLoad*. After minimizing the link energy we also need to check if maximum link load is infinite.

```
function [sol, Loads, maxLoad, energy] = ...
    HillClimbingStrategy(nNodes, Links, T, sP, nSP, sol, ...
    Loads, energy, L)
    nFlows = size(T,1);
    % set the best local variables
    maxLoad = max(max(Loads(:, 3:4)));
    bestLocalLoads = Loads;
    bestLocalSol = sol;
    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
                    [auxLoads, auxLinkEnergy] = ...
                        calculateLinkLoadEnergy(nNodes, ...
                        Links, T, sP, auxSol, L, 50);
                    nodeEnergy = calculateNodeEnergy(T, sP, ...
                        nNodes, 500, auxSol);
                    auxEnergy = auxLinkEnergy + nodeEnergy;

                    % check if the current link energy is ...
                        better then best
                    % local energy
                    if auxEnergy < bestLocalEnergy
                        bestLocalLoads = auxLoads;
                        bestLocalSol = auxSol;
                        bestLocalEnergy = auxEnergy;
                    end
                end
            end
        end

        if bestLocalEnergy < energy
            Loads = bestLocalLoads;
            sol = bestLocalSol;
            energy = bestLocalEnergy;

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

2.2 Exercise 2.b)

2.2.1 Code

For the exercise 2.b) we need to used the updated functions of *greedyRandomizedStrategy.m* and *HillClimbingStrategy.m*, we also need to change the comparison condition to check if the network energy consumption has improved, for that, we need to move the calculation of the energy consumed by the nodes inside the loop.

```
% Initial variables
load('InputDataProject2.mat');
nNodes = size(Nodes,1);
nFlows_uni = size(T_uni, 1);
anycastNodes = [5 12];

% Traffic flows for unicast service
% Computing up to k=1 shortest path for all flows
k = 2;
sP_uni = cell(1, nFlows_uni);            % sP{f}{i} is the ...
    i-th path of flow f
nSP_uni = zeros(1, nFlows_uni);          % nPS{f}{i} is the ...
    number of paths of flow f
for f = 1 : nFlows_uni
    [shortestPath, totalCost] = kShortestPath(L, ...
        T_uni(f,1), T_uni(f,2), k);
    sP_uni{f} = shortestPath;
    nSP_uni(f)= length(totalCost);
end
% Traffic flows for anycast service
[sP_any, nSP_any] = bestAnycastPaths(nNodes, anycastNodes, ...
    L, T_any);
% Reconstructing T matrix
% srcNode dstNode upRate dwRate
T_any = [T_any(:, 1) zeros(size(T_any,1), 1) T_any(:, 2:3)];
for i = 1 : size(T_any, 1)
    T_any(i, 2) = sP_any{i}{1}(end);
end
% Calculate general T, sP and nSP
T = [T_uni; T_any];
sP = cat(2, sP_uni, sP_any);
nSP = cat(2, nSP_uni, nSP_any);

t = tic;
timeLimit = 30;
bestLoad = inf;
bestEnergy = inf;
contador = 0;
while toc(t) < timeLimit
    % greedy randomzied start
    [sol, Loads, energy] = greedyRandomizedStrategy(nNodes, ...
        Links, T, sP, nSP, L);
```

```
    % The first solution should have a maxLinkLoad bellow ...
        the maxmium link
    % capacity
    while energy == inf
        [sol, Loads, energy] = ...
            greedyRandomizedStrategy(nNodes, Links, T, sP, ...
            nSP, L);
    end
    [sol, Loads, maxLoad, energy] = ...
        HillClimbingStrategy(nNodes, Links, T, sP, nSP, ...
        sol, Loads, energy, L);
    if energy < bestEnergy
        bestSol = sol;
        bestLoad = maxLoad;
        bestLoads = Loads;
        bestEnergy = energy;
        bestLoadTime = toc(t);
    end
    contador = contador + 1;
end

sleepingLinks = '';
for i = 1 : size(Loads, 1)
    if max(Loads(i, 3:4)) == 0
        sleepingLinks = append(sleepingLinks, ' {', ...
            num2str(Loads(i,1)), ', ', num2str(Loads(i,2)), ...
            '}');
    end
end

fprintf("E = %.2f \t W = %.2f \t No. sols = %d \t time = ...
    %.2f\n", bestEnergy, bestLoad, contador, bestLoadTime);
fprintf('List of links in sleeping mode:%s\n', sleepingLinks);
```

2.2.2 Results and Conclusions

Energy	Worst Link Load (Gbps)	No. Solutions	Time (s)
E = 775.64	W = 45.60	No. sols = 1372	time = 0.05

List of links in sleeping mode: {1, 2} {1, 7} {2, 3}

The results got on 1.a) and 1.b) were: worst link load of 49.90 Gbps and 851.81 of network energy consumption. In this case, we got a better value of energy consumption and worst link load, the value of energy is to be expected because our algorithm minimized the network energy consumption, the fact that we have a lower value of worst link load was by chance, however, with a k equal to 2, more paths are considered and it may be due to that.

On a side note, it is probably relevant that the list of links in sleeping mode might vary, due to the random factor of the algorithms, as referred on previous exercises. This occurs due to the fact that there may be different solutions that achieved the "best" energy consumption.

2.3 Exercise 2.c)

2.3.1 Code

The following code snippet is the same as it was shown in exercise 1.d) (except the value of k, which is 6, now).

```
% Initial variables
load('InputDataProject2.mat');
nNodes = size(Nodes,1);
nFlows_uni = size(T_uni, 1);
lc = 50;          % Link capacity of 50Gbps
nc = 500;         % Node capacity of 500Gbps
anycastNodes = [5 12];

% Traffic flows for unicast service
% Computing up to k=1 shortest path for all flows
k = 6;
sP_uni = cell(1, nFlows_uni);            % sP{f}{i} is the ...
    i-th path of flow f
nSP_uni = zeros(1, nFlows_uni);          % nPS{f}{i} is the ...
    number of paths of flow f
for f = 1 : nFlows_uni
    [shortestPath, totalCost] = kShortestPath(L, ...
        T_uni(f,1), T_uni(f,2), k);
    sP_uni{f} = shortestPath;
    nSP_uni(f)= length(totalCost);
end
% Traffic flows for anycast service
[sP_any, nSP_any] = bestAnycastPaths(nNodes, anycastNodes, ...
    L, T_any);

% Reconstructing T matrix
% srcNode dstNode upRate dwRate
T_any = [T_any(:, 1) zeros(size(T_any,1), 1) T_any(:, 2:3)];
for i = 1 : size(T_any, 1)
    T_any(i, 2) = sP_any{i}{1}(end);
end

% Calculate general T, sP and nSP
T = [T_uni; T_any];
sP = cat(2, sP_uni, sP_any);
nSP = cat(2, nSP_uni, nSP_any);

t = tic;
timeLimit = 120;
bestLoad = inf;
```

```
bestEnergy = inf;
contador = 0;
while toc(t) < timeLimit
    % greedy randomzied start
    [sol, Loads, energy] = greedyRandomizedStrategy(nNodes, ...
        Links, T, sP, nSP, L);

    % The first solution should have a maxLinkLoad bellow ...
        the maxmium link
    % capacity
    while energy == inf
        [sol, Loads, energy] = ...
            greedyRandomizedStrategy(nNodes, Links, T, sP, ...
            nSP, L);
    end

    [sol, Loads, maxLoad, energy] = ...
        HillClimbingStrategy(nNodes, Links, T, sP, nSP, ...
        sol, Loads, energy, L);

    if energy < bestEnergy
        bestSol = sol;
        bestLoad = maxLoad;
        bestLoads = Loads;
        bestEnergy = energy;
        bestLoadTime = toc(t);
    end
    contador = contador + 1;
end

sleepingLinks = '';
for i = 1 : size(Loads, 1)
    if max(Loads(i, 3:4)) == 0
        sleepingLinks = append(sleepingLinks, ' {', ...
            num2str(Loads(i,1)), ', ', num2str(Loads(i,2)), ...
            '}');
    end
end

fprintf("E = %.2f \t W = %.2f \t No. sols = %d \t time = ...
    %.2f\n", bestEnergy, bestLoad, contador, bestLoadTime);
fprintf('List of links in sleeping mode:%s\n', sleepingLinks);
```

2.3.2 Results and Conclusions

Energy	Worst Link Load (Gbps)	No. Solutions	Time (s)
E = 700.93	W = 48.00	No. sols = 900	time = 47.87

List of links in sleeping mode: {1, 2} {1, 7} {2, 3} {13, 14}

Increasing the value of k from 2 to 6 in the shortest path calculations has resulted in a lower network energy consumption, and in a higher worst link load.

Generally, increasing the value of k allows the algorithm to explore more paths of each flow, which can potentially lead to better solutions in terms of lower network energy consumption. In this particular case, it is possible that the paths with the lowest energy consumption have an higher maximum link load. As a result, the algorithm may prioritize minimizing the energy consumption over minimizing the maximum link load, leading to a solution with a lower energy consumption but a higher maximum link load.

Like in the previous exercises, it is probably important to note that the list of sleeping links oscillates, due to the same reasons stated before: random factor of the algorithm which may lead to different solutions with same the energy consumption values.

We have also to take in consideration that the values of network energy consumption fluctuate, this is probably because there are a lot of candidate solutions for a good network energy consumption within the time limit.

Lastly, since the time to get the best solution would get near the maximum value defined (30 seconds), we decided to increase it to 60 seconds. We repeated this process until the time it took to find the solution with the lowest energy consumption wasn't near the time limit, which occurred with the time limit variable equal to 120 seconds.

2.4 Exercise 2.d)

2.4.1 Results and Conclusions

We can see that the solution in exercise 2.c) has a lower energy consumption but a higher maximum link load compared to the solution in exercise 1.e). This is to be expected since that the algorithm in exercise 2.c) is using more energy-efficient paths while trading the maximum link load of the network, this means that, for this network and the algorithm used, the paths that minimize the network energy consumption result in a higher maximum link load.

Regarding the performance of the algorithms, we can see that exercise 1.e) takes less time to find a solution (0.03 seconds) compared to exercise 2.c) (47.87 seconds). This is because we let 2.c) runs for longer to get the "best solution", the fact that energy optimization takes longer than the maximum link load optimization is likely due to the fact that the energy optimization algorithm does more calculations to decide if the algorithm has improved, *calculateLinkLoadEnergy.m* finishes both at the same time but to decide if the energy has improved we have to also execute *calculateNodeEnergy.m*.

In task 1.e), the number of solutions created is 1948, while in task 2.c), the number of solutions created is 900. This means that task 2.c) is creating fewer solutions than task 1.e), which could be due to a number of factors. One possibility is that the greedy randomized strategy used in task 1.e) is more efficient at finding good solutions, or that the Hill climbing strategy used in task 1.e) is more efficient at improving upon the solutions found by the greedy randomized strategy. Alternatively, it could be that the optimization objectives in task 1.e) are more straightforward or easier to optimize than for those in task 2.c), leading to a fewer number of solutions that need to find an optimal solution.

Overall, the difference in the results between the two exercises can be attributed to the different optimization targets and the inherent trade-offs between maximizing network resource utilization and minimizing energy consumption.

Chapter 3

Task 3

3.1 Exercise 3.a)

3.1.1 Aprroach

For this task our approach was letting the algorithm run with no restriction on the maximum link load, and then the only restriction is that the maximum link load shouldn't exceed 100Gbps. When we get the final result we will see which links need to have a capacity of 100Gbps, which is done by seeing what are the loads each link needs to support based on the best solution achieved.

3.1.2 Code

For Task 3, *greedyRandomizedStrategy.m* and *HillClimbingStrategy.m* haven't changed, what changed was *calculateLinkLoadEnergy.m*, therefore we called the function in a different way. We changed calculateLinkLoadEnergy to set the energy to inf when the maximum link load exceeds 100Gbps, this way, we can prevent that the algorithm doesn't pick this solution. Another change was that the *linkEnergy* equation was chosen based on the current maximum link load of the link we are iterating.

```
function [Loads, linkEnergy] = ...
    calculateLinkLoadEnergy(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)];
    linkEnergy = 0;
    for i= 1:nLinks
        Loads(i,3)= aux(Loads(i,1),Loads(i,2));
        Loads(i,4)= aux(Loads(i,2),Loads(i,1));

        maxLoad = max(max(Loads(:, 3:4)));
        % If the worst link load is greater than max ...
            capacity , energy will be infinite
        if maxLoad > 100
            linkEnergy = inf;
        else
            maxCurrentLoad = max(Loads(i, 3:4));
            % link in sleeping mode
            if maxCurrentLoad == 0
                linkEnergy = linkEnergy + 2;        % El = ...
                    2 whatever the link capacity
            else
                % len from nodeA to nodeB
                len = L(Loads(i, 1), Loads(i, 2));

                % energy calculation dependent of link capacity
                if maxCurrentLoad ≤ 50
                    linkEnergy = linkEnergy + 6 + 0.2 * len;
                elseif maxCurrentLoad > 50
                    linkEnergy = linkEnergy + 8 + 0.3 * len;
                else
                    fprintf('Error: Link capacity is ...
                        %.2f\n', maxCurrentLoad);
                end
            end
        end
    end
end
```

Therefore, we need to change *greedyRandomizedStrategy.m* and *HillClimbingStrategy.m* to call *calculateLinkLoadEnergy.m* like this:

```
function [sol, Loads, energy] = ...
    greedyRandomizedStrategy(nNodes, Links, T, sP, nSP, L)
    Loads = inf;
    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_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
            [Loads, linkEnergy] = ...
                calculateLinkLoadEnergy(nNodes, Links, T, ...
                sP, sol, L);
            if linkEnergy < inf
                nodeEnergy = calculateNodeEnergy(T, sP, ...
                    nNodes, 500, sol);
                energy = linkEnergy + nodeEnergy;
            else
                energy = inf;
            end

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

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

```
function [sol, Loads, maxLoad, energy] = ...
    HillClimbingStrategy(nNodes, Links, T, sP, nSP, sol, ...
    Loads, energy, L)
    nFlows = size(T,1);
    % set the best local variables
    maxLoad = max(max(Loads(:, 3:4)));
    bestLocalLoads = Loads;
    bestLocalSol = sol;
    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
                    [auxLoads, auxLinkEnergy] = ...
                        calculateLinkLoadEnergy(nNodes, ...
                        Links, T, sP, auxSol, L);
                    nodeEnergy = calculateNodeEnergy(T, sP, ...
                        nNodes, 500, auxSol);
                    auxEnergy = auxLinkEnergy + nodeEnergy;

                    % check if the current link energy is ...
                        better then best
                    % local energy
                    if auxEnergy < bestLocalEnergy
                        bestLocalLoads = auxLoads;
                        bestLocalSol = auxSol;
                        bestLocalEnergy = auxEnergy;
                    end
                end
            end
        end

        if bestLocalEnergy < energy
            Loads = bestLocalLoads;
            sol = bestLocalSol;
            energy = bestLocalEnergy;

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

3.2 Exercise 3.b)

3.2.1 Code

There isn't any particular big change in the following code when comparing it to the previous exercise. We now use k equal to 6 and *timeLimit* of 60 seconds. One thing that we added was checking what are the links that changed the capacity from 50Gbps to 100Gbps by verifying (on the final solution) if the maximum load on a link surpasses 50Gbps.

```
% Initial variables
load('InputDataProject2.mat');
nNodes = size(Nodes,1);
nFlows_uni = size(T_uni, 1);
anycastNodes = [5 12];

% Traffic flows for unicast service
% Computing up to k=1 shortest path for all flows
k = 6;
sP_uni = cell(1, nFlows_uni);           % sP{f}{i} is the ...
    i-th path of flow f
nSP_uni = zeros(1, nFlows_uni);         % nPS{f}{i} is the ...
    number of paths of flow f
for f = 1 : nFlows_uni
    [shortestPath, totalCost] = kShortestPath(L, ...
        T_uni(f,1), T_uni(f,2), k);
    sP_uni{f} = shortestPath;
    nSP_uni(f)= length(totalCost);
end
% Traffic flows for anycast service
[sP_any, nSP_any] = bestAnycastPaths(nNodes, anycastNodes, ...
    L, T_any);

% Reconstructing T matrix
% srcNode dstNode upRate dwRate
T_any = [T_any(:, 1) zeros(size(T_any,1), 1) T_any(:, 2:3)];
for i = 1 : size(T_any, 1)
    T_any(i, 2) = sP_any{i}{1}(end);
end

% Calculate general T, sP and nSP
T = [T_uni; T_any];
sP = cat(2, sP_uni, sP_any);
nSP = cat(2, nSP_uni, nSP_any);

t = tic;
timeLimit = 60;
bestLoad = inf;
bestEnergy = inf;
contador = 0;
while toc(t) < timeLimit
    % greedy randomzied start
```

```
    [sol, Loads, energy] = greedyRandomizedStrategy(nNodes, ...
        Links, T, sP, nSP, L);

    % The first solution should have a maxLinkLoad bellow ...
        the maxmium link
    % capacity
    while energy == inf
        [sol, Loads, energy] = ...
            greedyRandomizedStrategy(nNodes, Links, T, sP, ...
            nSP, L);
    end

    [sol, Loads, maxLoad, energy] = ...
        HillClimbingStrategy(nNodes, Links, T, sP, nSP, ...
        sol, Loads, energy, L);

    if energy < bestEnergy
        bestSol = sol;
        bestLoad = maxLoad;
        bestLoads = Loads;
        bestEnergy = energy;
        bestLoadTime = toc(t);
    end
    contador = contador + 1;
end

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

sleepingLinks = '';
for i = 1 : size(Loads, 1)
    if max(Loads(i, 3:4)) == 0
        sleepingLinks = append(sleepingLinks, ' {', ...
            num2str(Loads(i,1)), ', ', num2str(Loads(i,2)), ...
            '}');
    end
end

fprintf("E = %.2f \t W = %.2f \t No. sols = %d \t time = ...
    %.2f\n", bestEnergy, bestLoad, contador, bestLoadTime);
fprintf("List of links that changed to 100Gbps:%s\n", ...
    changedLinks);
fprintf('List of links in sleeping mode:%s\n', sleepingLinks);
```

3.2.2 Results and Conclusions

In this exercise, all the values are constant except the list of links in sleeping mode that fluctuates. This is due to the same reasons enunciated before. Since the algorithms have a random factor and this leads to different solutions with same levels of energy consumption, while aiming for the lowest energy consumption.

Comparing with the values of 2.c) (energy of 700.93; worst link load of 48 Gbps; 900 solutions; 47.87 seconds with a limit of 120 seconds), we can assess that, since the aim of both the algorithms (exercise 2.c) and this exercise) is to minimize the value of the energy consumption and the fact that we get a lower energy consumption on this exercise and a worse maximum link load, is due to the fact that we have the possibility to reach even worse maximum link load values (100 Gbps), therefore, since we have a bigger trade-off to offer, the value of energy consumption reached is lower. This is noticeable in this exercise's worst link load value: 80.30 Gbps.

In conclusion, this is the key difference between the two exercises. A worst maximum link load may be reached, thus, there are more possible paths for a better energy consumption value that would reach higher maximum link load values than 50 Gbps.

Energy	Worst Link Load (Gbps)	No. Solutions	Time (s)
678.80	80.30	1027	1.17

List of links that changed to 100Gbps: {3, 4}

List of links in sleeping mode: {1, 2} {1, 7} {2, 3} {6, 8} {10, 11} {11, 13} {13, 14}

3.3 Exercise 3.c)

3.3.1 Code

On this exercise we changed *bestAnycastPaths.m* and *calculateNodeEnergy.m* to consider cases when the destination anycast node can also be a source of a flow.

On *bestAnycastPaths.m* we first check if the node is a destination anycast node, and if it is we also check if that node is a member of T_any. If this happens, the node in question is simultaneously a source and destination node of the anycast service, if that is the case we automatically select it as the shortest path.

```
function [sP, nSP] = bestAnycastPaths(nNodes, anycastNodes, ...
    L, T_any)
    sP = cell(1, nNodes);
    nSP = zeros(1, nNodes);
    for n = 1:nNodes
        if ismember(n, anycastNodes)    % if the node is a ...
            anycastNode skip it
            if ismember(n, T_any(:, 1))
                sP{n} = {[n n]};
                nSP(n) = 1;
            else
                nSP(n) = -1;
            end
            continue;
        end

        if ¬ismember(n, T_any(:, 1))     % node is not from ...
            T_any matrix
            nSP(n) = -1;
            continue;
        end

        best = inf;
        for a = 1:length(anycastNodes)
            [shortestPath, totalCost] = kShortestPath(L, n, ...
                anycastNodes(a), 1);

            if totalCost(1) < best
                sP{n} = shortestPath;
                nSP(n) = length(totalCost);
                best = totalCost;
            end
        end
    end

    nSP = nSP(nSP≠-1);                      % remove unwanted values
    sP = sP(¬cellfun(@isempty, sP));    % remove empty ...
        entry from the cell array
end
```

On *calculateNodeEnergy.m* we have to see the case where the first node of the shortest path is equal to the last node, this only happens when the anycast node has itself as the destination, in this case we don't change the *nodesTraffic* variable.

```
function energy = calculateNodeEnergy(T, sP, nNodes, nc, sol)
    nodesTraffic = zeros(1, nNodes);

    for flow = 1 : size(T,1)
        if sol(flow) ≠ 0
            nodes  = sP{flow}{sol(flow)};
```

```

            if nodes(1) == nodes(end)
                continue;
            end

            for n = nodes
                nodesTraffic(n) = nodesTraffic(n) + ...
                    sum(T(flow, 3:4));
            end
        end
    end

    energy = sum(10 + 90 * (nodesTraffic/nc).^2);
end
```

On this exercise we do a list of possible pairs of destination anycast nodes and test all of them with the generic algorithm performed in the previous exercise.

```
% Initial variables
load('InputDataProject2.mat');
nNodes = size(Nodes,1);
nFlows_uni = size(T_uni, 1);
anycastNodesList = nchoosek([4 5 6 12 13], 2);

% Traffic flows for unicast service
% Computing up to k=1 shortest path for all flows
k = 6;
sP_uni = cell(1, nFlows_uni);            % sP{f}{i} is the ...
    i-th path of flow f
nSP_uni = zeros(1, nFlows_uni);          % nPS{f}{i} is the ...
    number of paths of flow f
for f = 1 : nFlows_uni
    [shortestPath, totalCost] = kShortestPath(L, ...
        T_uni(f,1), T_uni(f,2), k);
    sP_uni{f} = shortestPath;
    nSP_uni(f)= length(totalCost);
end

bestEnergy = inf;
contador = 0;
for pair = 1:size(anycastNodesList, 1)
    anycastNodes = anycastNodesList(pair, :);
    % Traffic flows for anycast service
    [sP_any, nSP_any] = bestAnycastPaths(nNodes, ...
        anycastNodes, L, T_any);

    % Reconstructing T matrix
    % srcNode dstNode upRate dwRate
    T_any2 = [T_any(:, 1) zeros(size(T_any,1), 1) T_any(:, ...
        2:3)];
    for i = 1 : size(T_any, 1)
```

```
        T_any2(i, 2) = sP_any{i}{1}(end);
    end

    % Calculate general T, sP and nSP
    T = [T_uni; T_any2];
    sP = cat(2, sP_uni, sP_any);
    nSP = cat(2, nSP_uni, nSP_any);

    t = tic;
    timeLimit = 60;
    while toc(t) < timeLimit
        % greedy randomzied start
        [sol, Loads, energy] = ...
            greedyRandomizedStrategy(nNodes, Links, T, sP, ...
            nSP, L);
        % The first solution should have a maxLinkLoad ...
            bellow the maxmium link
        % capacity
        while energy == inf
            [sol, Loads, energy] = ...
                greedyRandomizedStrategy(nNodes, Links, T, ...
                sP, nSP, L);
        end

        [sol, Loads, maxLoad, energy] = ...
            HillClimbingStrategy(nNodes, Links, T, sP, nSP, ...
            sol, Loads, energy, L);

        if energy < bestEnergy
            bestSol = sol;
            bestLoad = maxLoad;
            bestLoads = Loads;
            bestEnergy = energy;
            bestLoadTime = toc(t);
            bestNodes = anycastNodes;
        end
        contador = contador + 1;
    end
end

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

sleepingLinks = '';
for i = 1 : size(Loads, 1)
    if max(Loads(i, 3:4)) == 0
        sleepingLinks = append(sleepingLinks, ' {', ...
            num2str(Loads(i,1)), ', ', num2str(Loads(i,2)), ...
```

```
                '}');
        end
    end

    fprintf('The best pair of nodes: %s\n', num2str(bestNodes));
    fprintf("E = %.2f \t W = %.2f \t No. sols = %d \t time = ...
        %.2f\n", bestEnergy, bestLoad, contador, bestLoadTime);
    fprintf("List of links that changed to 100Gbps:%s\n", ...
        changedLinks);
    fprintf('List of links in sleeping mode:%s\n', sleepingLinks);
```

3.3.2 Results and Conclusions

Firstly, the values are constant, but like the other exercises the list of sleeping links may vary, because of the same reasons explained before - there is a random factor associated to the algorithms and this leads to different solutions with same levels of energy consumption, while aiming for the lowest energy consumption.

When comparing the values to exercise 3.b), where we got an energy consumption of 678.80, a worst link load value of 80.30 Gbps, a number of solutions 1027 and 1.17 seconds to get the best solution, we conclude that, in this exercise (part c of exercise 3) there's a better (lower) energy consumption value. This happens as a result of using more possible nodes and it is possible that the node goes from him to himself, which won't add energy consumption. In exercise 3.b) a node to go to itself would need to do a certain path and, therefore, consume more energy. All of this is due to the way of how we calculate the best energy.

The best pair of nodes: 4 13

Energy	Worst Link Load (Gbps)	No. Solutions	Time (s)
655.18	61.90	8805	5.41

List of links that changed to 100Gbps: {13, 14}

List of links in sleeping mode: {1, 2} {1, 7} {2, 3} {3, 6} {5, 7} {8, 11} {13, 14}

3.3.3 Different Approach

In the approach above, when a anycast node can be a source and destination node at the same time, we do not add any value to the energy consumption of the network, this approach could be wrong and we decided to test another approach, if the node is simulaneously a source and destination node of anycast flow we add the traffic only one time.

The resulting *calculateNodeEnergy.m* should look like this.

```
function energy = calculateNodeEnergy(T, sP, nNodes, nc, sol)
    nodesTraffic = zeros(1, nNodes);

    for flow = 1 : size(T,1)
        if sol(flow) ≠ 0
            nodes  = sP{flow}{sol(flow)};

            if nodes(1) == nodes(end)
                nodesTraffic(nodes(1)) = ...
                    nodesTraffic(nodes(1)) + sum(T(flow, 3:4));
                continue;
            end

            for n = nodes
                nodesTraffic(n) = nodesTraffic(n) + ...
                    sum(T(flow, 3:4));
            end
        end
    end

    energy = sum(10 + 90 * (nodesTraffic/nc).^2);
end
```

And, when running the exercise again with this function we get the following results:

The best pair of nodes: 4 13

Energy	Worst Link Load (Gbps)	No. Solutions	Time (s)
658.43	61.90	8305	5.41

List of links that changed to 100Gbps: {13, 14}

List of links in sleeping mode: {1, 2} {1, 7} {2, 3} {3, 6} {13, 14}

This modification isn't significant to the best solution achieved, only a different value of total energy consumption of the network.

Chapter 4

Information

The contributions between each member of the group were equal.
The project's repository can be viewed here: `https://github.com/PedroRocha9/MDRS`
A collaborator invitation in GitHub was sent to the professor's organization email: asou@ua.pt.
The function *kShortestPath.m* is the same used in the practical classes.