MDRS Mini-Project 2

This is a report the Second Mini-Project of MDRS university class, entitled **Traffic Engineering of Telecommunication Networks**. The work carried out was conducted by:

- Gonçalo Silva, 103244 - Participation 50%
- Catarina Barroqueiro, 103895 - Participation 50%

Exercise 1.a

Code

This exercise evaluates the feasibility of a routing solution where all flows are routed through the path with the minimum propagation delay in the network. The script loads input data and calculates key network characteristics, including propagation delay, throughput, and energy consumption for routers and links.

The script undertakes the following steps:

1. **Propagation Delay Calculation**: The propagation delay on each direction of every link is computed using the formula ($D = \frac{L}{v}$) , where (D) is the propagation delay matrix, (L) is the link length matrix, and (v) is the speed of light in fibers.

2. **Total Throughput Traffic on Each Link**: The script calculates the total throughput traffic on each link by summing the throughput values for all flows traversing the respective links. This is represented by the matrix ($total\_throughput\_per\_link$) .

3. **Router and Link Capacity Checks**: The script checks the total throughput traffic against the capacities of both routers and links. The router's energy consumption model is ($20 + 80 \times \sqrt{\frac{t}{T}}$), where (t) is the total throughput traffic supported by the router, and (T) is the router's total capacity. The link's energy consumption model is ($9 + 0.3 \times l$), where (l) is the length of the link.

The feasibility of the routing solution is determined by checking whether the calculated total throughput exceeds the capacities of routers or links.

```
%% Exercise 1.a)
clear all
close all
clc

% Load input data
load('InputDataProject2.mat')

% Assuming T1 is the matrix for Service 1
T = [T1; T2];
nFlows = size(T, 1);

% Compute the propagation delay matrix D
v = 2e5; % speed of light on fibers in km/sec
D = L / v;

% Computing up to k=2 link disjoint paths for all flows
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

% Check the feasibility by considering the load on nodes
sol = ones(1, nFlows); % All flows follow the shortest paths
[Loads, unusedLinks] = calculateLinkLoads(size(Nodes, 1), Links, T, sP,
sol);

% Check if the solution is feasible
isFeasible = all(nSP >= 2) && all(all(Loads(:, 3:4) <= 1000));

% Check if node loads are within capacity
nodeLoads = sum(Loads(:, 3:4), 1); % Sum of loads for each node
isFeasible = isFeasible && all(nodeLoads <= 1000);

% Display the results
if isFeasible
    fprintf('The solution exists:\n');
    for f = 1:nFlows
```

```
        fprintf('\- tFlow %d (%d -> %d): Paths = ', f, T(f, 1), T(f, 2));
        for p = 1:k
            fprintf('%s ', num2str(sP{f}{p}));
        end
        fprintf('\n');
    end
    fprintf("\n");

    % Calculate and display the worst link load
    worstLinkLoad = max(max(Loads(:, 3:4)));
    if worstLinkLoad <= 100 % max link capacity
        fprintf('The solution is feasible. All traffic flows within
capacity\n');
    else
        fprintf('The solution is not feasible, since Worst Link Load of %.2f
exceeds maximum link capacity of 100 Gbps\n', worstLinkLoad);
    end
    fprintf('\n');

    % Call the print_solution_stats function
    print_solution_stats(Loads, calculateNetworkEnergyConsumption(Loads, L,
T, sol, sP), 0, 0, sol, sP, T1, T2, D);
else
    fprintf('The solution does not exist. Some flows are unreachable\n');
end
```

Results

The analysis of the solution where all flows are routed through the path with the minimum propagation delay of the network yields the following statistics:

```
- Worst Link Load: 142.80 Gbps
- Average Upload Link Load: 34.46 Gbps
- Average Download Link Load: 34.49 Gbps
- Network Energy Consumption: 2829.25
- Average Round-Trip Time:
  - Service 1: 5.039 ms
  - Service 2: 5.611 ms
- Number of Links Without Traffic: 4
- Links with No Traffic: {1,7} {3,8} {6,15} {12,13}
- Number of Cycles: 0
- Best Solution Time: 0.00
```

Feasibility Analysis

The feasibility of the solution is assessed based on the obtained results. The worst link load, registering at 142.80 Gbps, exceeds the 100 Gbps capacity of the links. This violation of the capacity constraint renders the solution impractical.

While the average upload and download link loads appear balanced, the presence of excessively loaded links, as indicated by the worst link load, undermines the overall feasibility.

The calculated network energy consumption of 2829.25, though not a determining factor, should be considered in the broader context of network optimization.

The average round-trip times for both services (Service 1: 5.039 ms, Service 2: 5.611 ms) surpass acceptable limits, negatively impacting the quality of service and contributing to the overall infeasibility of the solution.

The identification of links without traffic does not directly impact feasibility but provides insights for potential optimization.

The absence of cycles in the network structure is a positive factor, suggesting a tree-like routing solution that contributes favorably to network efficiency.

Conclusion

The solution is deemed infeasible primarily due to the violation of link capacity constraints and elevated round-trip times. The worst link load exceeding the link capacity renders the current routing solution impractical. Further optimization strategies, such as load balancing and alternative routing paths, should be explored to ensure a feasible and efficient network configuration.

Exercise 1.b

Code

Exercise 1.b aims to minimize the worst link load in an MPLS network by developing a Multi-Start Hill Climbing algorithm with initial Greedy Randomized solutions.

Multi-Start Hill Climbing

Multi-Start Hill Climbing is a metaheuristic algorithm used for optimization. In this specific case, it is applied to refine the solutions generated by the Greedy Randomized Strategy. The algorithm iteratively explores the solution space, making local improvements to find an optimal or near-optimal solution.

Greedy Randomized Strategy

The Greedy Randomized Strategy is employed to solve combinatorial optimization problems, particularly focusing on minimizing the worst link load in a symmetrical single-path routing solution for multiple services. This iterative algorithm generates solutions by probabilistically selecting optimal choices.

Multi-Start Hill Climbing with Greedy Randomized Strategy

The Multi-Start Hill Climbing with Greedy Randomized Strategy is employed to solve combinatorial optimization problems, particularly focusing on minimizing the worst link load in a symmetrical single-path routing solution for multiple services. This iterative algorithm generates solutions by probabilistically selecting optimal choices.

The `multiStartHillClimbingGreedy` function is the implementation of this strategy. It aims to iteratively enhance routing solutions within a time limit, utilizing a randomized approach to minimize the worst link load.

Input Parameters:

- `sP`: Shortest paths for each flow and service.
- `nSP`: Number of shortest paths for each flow.
- `T`: Service flows matrix.
- `nNodes`: Number of nodes in the network.
- `Links`: Link information matrix.
- `timeLimit`: Time limit for the algorithm.

Output:

- `bestSol`: Best routing solution.
- `bestLoads`: Link loads corresponding to the best solution.
- `bestLoad`: Worst link load in the best solution.
- `contador`: Number of iterations.
- `somador`: Accumulated link loads during iterations.
- `bestLoadTime`: Time taken for the best solution.

```
function [bestSol, bestLoads, bestLoad, contador, somador, bestLoadTime] =
multiStartHillClimbingGreedy(sP, nSP, T, nNodes, Links, timeLimit)
    nFlows = height(T);
    t= tic;
    bestLoad= inf;  % objective function value
    contador= 0;
    somador= 0;

    while toc(t) < timeLimit
        % Generate initial solution using Greedy Randomized approach
        sol= zeros(1,nFlows);
        for f= randperm(nFlows)
            auxBest= inf;
            for i=1:nSP(f)
                sol(f)= i;
                auxLoads= calculateLinkLoads(nNodes,Links,T,sP,sol);
                auxLoad= max(max(auxLoads(:,3:4)));
                if auxLoad < auxBest
                    auxBest= auxLoad;
                    ibest=i;
                end
            end
            sol(f)= ibest;
        end
        Loads= calculateLinkLoads(nNodes,Links,T,sP,sol);
        load= max(max(Loads(:,3:4)));   % calculate max capacity of link
loads

        % Perform hill climbing on the Greedy Randomized initial solution
        [sol, load] = hillClimbing(sP, nSP, T, nNodes, Links, sol, load);

        % Update the best solution if a better one is found
        if load < bestLoad
            bestSol = sol;
            bestLoad = load;
            bestLoads = Loads;
            bestLoadTime= toc(t);
        end

        contador = contador + 1;
        somador = somador + load;
    end
end
```

`calculateLinkLoads` Function

The `calculateLinkLoads` function plays a crucial role in determining link loads within a routing solution and identifying unused links. It iterates through the flows and paths in the given solution, updating the auxiliary matrix to calculate load distribution.

```
function [Loads, unusedLinks] = calculateLinkLoads(nNodes, Links, T, sP,
Solution)
    nFlows = size(T, 1);
    nLinks = size(Links, 1);
    aux = zeros(nNodes);

    for i = 1:nFlows
        if Solution(i) > 0
            path = sP{i}{Solution(i)};
            for j = 2:length(path)
                aux(path(j-1), path(j)) = aux(path(j-1), path(j)) + T(i, 3);
                aux(path(j), path(j-1)) = aux(path(j), path(j-1)) + T(i, 4);
            end
        end
    end

    Loads = [Links zeros(nLinks, 2)];
    for i = 1:nLinks
        Loads(i, 3) = aux(Loads(i, 1), Loads(i, 2));
        Loads(i, 4) = aux(Loads(i, 2), Loads(i, 1));
    end

    % Identify unused links
    unusedLinks = find(all(Loads(:, 3:4) == 0, 2));
end
```

The script in the file ex1_b.m for Exercise 1.b efficiently addresses the optimization problem of computing a symmetrical single-path routing solution for both services, aiming to minimize the worst link load. The script loads input data, calculates the k-shortest paths using the provided kShortestPath algorithm, and executes the Multi-Start Hill Climbing algorithm with the Greedy Randomized Strategy. It displays key results such as the best solution, worst link load, and the algorithm's convergence time.

```
%% Exercise 1.b)
clear all
close all
clc

% Load input data
load('InputDataProject2.mat')

% Assuming T1 is the matrix for Service 1
T = [T1; T2];
nFlows = size(T, 1);

% Compute the propagation delay matrix D
v = 2e5; % speed of light on fibers in km/sec
D = L / v;

% Computing up to k=2 link disjoint paths for all flows
```

```
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

% Check the feasibility by considering the load on nodes
sol = ones(1, nFlows); % All flows follow the shortest paths
[Loads, unusedLinks] = calculateLinkLoads(size(Nodes, 1), Links, T, sP,
sol);

% Check if the solution is feasible
isFeasible = all(nSP >= 2) && all(all(Loads(:, 3:4) <= 1000));

% Display the non-greedy results
if isFeasible
    for f = 1:nFlows
        fprintf('Flow %d (%d -> %d): Paths = ', f, T(f, 1), T(f, 2));
        for p = 1:k
            fprintf('%s ', num2str(sP{f}{p}));
        end
        fprintf('\n');
    end

    % Calculate and display the worst link load
    worstLinkLoad = max(max(Loads(:, 3:4)));

    % Multi-Start Hill Climbing
    fprintf('Multi-Start Hill Climbing with Greedy Randomized Results:\n');
    [bestSol, bestLoads, bestLoad, contador, somador, bestLoadTime] =
multiStartHillClimbingGreedy(sP, nSP, T, size(Nodes, 1), Links, 60);

    % Call the print_solution_stats function with energy consumption
    print_solution_stats(bestLoads,
calculateNetworkEnergyConsumption(bestLoads, L, T, bestSol, sP), contador,
bestLoadTime, bestSol, sP, T1, T2, D);
else
    fprintf('The solution is not feasible. Some flows may be unreachable or
node loads exceed capacity.\n');
end
```

Exercise 1.c

Executing our algorithm for 60 seconds with *k=2* resulted in the following output:

```
Worst link load: 87.80 Gbps
Average Upload link load: 38.01 Gbps
Average Download link load: 37.25 Gbps
Network energy consumption: 2847.95
Average Round-trip time:
  - Service 1: 5.447 ms
  - Service 2: 5.633 ms
Number of links without traffic: 4
Links with no traffic: {1,5} {1,7} {6,15} {12,13}
Number of cycles: 64264
Best solution time: N/A
```

Exercise 1.d

Executing our algorithm for 60 seconds with *k=6* resulted in the following output:

```
Worst link load: 76.40 Gbps
Average Upload link load: 40.84 Gbps
Average Download link load: 40.63 Gbps
Network energy consumption: 2931.56
Average Round-trip time:
  - Service 1: 6.882 ms
  - Service 2: 6.010 ms
Number of links without traffic: 3
Links with no traffic: {3,6} {6,15} {12,13}
Number of cycles: 12108
Best solution time: 3.93
```

Analysis and Conclusions

The comparison between Exercise 1.c and Exercise 1.d yields insightful observations on the network's performance and efficiency.

Exercise 1.d achieved a lower worst link load (76.40 Gbps) compared to Exercise 1.c (87.80 Gbps), signifying an improvement in network utilization. However, the average upload and download link loads in Exercise 1.d were higher, suggesting a more balanced distribution of traffic.

Network energy consumption increased in Exercise 1.d (2931.56) compared to Exercise 1.c (2847.95), potentially attributed to the elevated network load. This increase in energy consumption should be considered in the context of overall network optimization.

Round-trip times for both services were higher in Exercise 1.d, indicating a potential trade-off between network performance and load balancing.

Exercise 1.d reduced the number of links without traffic from 4 to 3, reflecting improved resource utilization.

The number of cycles required for Exercise 1.d (12108) was lower than Exercise 1.c (64264), suggesting that a higher value of k (number of candidate paths) leads to faster convergence to a solution.

In terms of solution time, Exercise 1.d achieved its best solution in a shorter time (3.93 seconds) compared to Exercise 1.c, highlighting the efficiency of the algorithm with a higher k value. These findings contribute to a comprehensive understanding of the network's behavior and aid in making informed decisions for further optimization.

Conclusion

The increased value of k in Exercise 1.d led to a more optimized solution in terms of link load distribution, albeit with slightly higher energy consumption and round-trip times. The trade-off between load balancing and network performance should be considered in selecting the appropriate k value. The algorithm's efficiency is evident in the reduced number of cycles and faster convergence to a solution in Exercise 1.d.

Exercise 2.a

The Multi-Start Hill Climbing algorithm in task 1 aimed to minimize the **Worst Link Load (WLL)**. However, in this exercise, the algorithm's objective changed to prioritize reducing energy costs while ensuring the maximum link load target of 100 Gbps isn't exceeded. This required changes in the algorithm's functions, particularly in the calculation of link and node energy costs.

The revised function focuses first on evaluating the energy consumption of links by assessing their throughput. Then, it proceeds to analyze the load on each node. This involves creating a matrix to represent each node and iteratively assign throughput values while calculating the energy costs associated with all nodes. The function can be found below:

```
function consumption= calculateEnergyConsumption(Lengths, Loads, T, sol, sP)
    % sP are the k shortest paths
    % sol is the solution with the computed shortest path
    consumption= 0;
    auxLoads= Loads';

    % Processing links
    for link=auxLoads
        if sum(link(3:4)) == 0 % Link in sleeping mode
            consumption= consumption + 2;
        else % Link in operation
            distance= Lengths(link(1), link(2));
            consumption= consumption + (9 + 0.3 * distance);
        end
    end

    % Processing nodes
    nNodes= length(Lengths);
    totalThroughput= zeros(nNodes,1);
    nFlows= height(T);
    capacity= 1000; % in Gbps

    for i= 1:nFlows
        if sol(i)>0
            path= sP{i}{sol(i)};
            throughput= T(i,3) + T(i,4);
            for node=path
                totalThroughput(node,1) = totalThroughput(node,1) +
throughput;
            end
        end
    end

    for i=1:length(totalThroughput(:,1))
        t = totalThroughput(i)/capacity;
        consumption = consumption + (20 + 80 * sqrt(t));
    end

end
```

This updated function revises the energy cost calculation process to prioritize minimizing energy cost while considering the link throughput and node loads within the network.

The Multi-Start algorithm function suffered changes, particularly in the initial calculation of Greedy Solutions. A while loop was introduced to handle situations where a solution isn't immediately obtained. The condition `if auxLoad <= (alpha * C)` involves two variables: alpha, which can be adjusted for potential enhancements in minimizing the **Worst Link Load (WLL)**, and **C**, representing the maximum capacity of the Link. All links have a scalar maximum load of *100 Gbps*. This condition ensures that only feasible solutions, within equipment constraints, are explored. Below is the adjusted code of the function:

```
(...)

while toc(t) < timeLimit
    continuar= true;
    % Respect alpha
    while continuar
        continuar= false;
        % Generate initial solution using Greedy Randomized approach
        sol= zeros(1,nFlows);
        for f= randperm(nFlows)
            ibest= 0;
            auxBestEnergy= inf;
            for i=1:nSP(f)
                sol(f)= i;
                auxLoads= calculateLinkLoads(nNodes,Links,T,sP,sol);
                auxLoad= max(max(auxLoads(:,3:4)));
                % assuming equal link capacity in each direction
                if auxLoad <= (alpha * C)
                    auxenergy= calculateEnergyConsumption(L, auxLoads, T,
sol, sP);
                    if  auxenergy < auxBestEnergy
                        auxBestEnergy= auxenergy;
                        ibest= i;
                    end
                end
            end
            if ibest > 0
                sol(f)= ibest;
            else
                continuar= true;
                break;
            end
        end
    end
    Loads= calculateLinkLoads(nNodes,Links,T,sP,sol);
    energy= calculateEnergyConsumption(L, Loads, T, sol, sP);

    % Perform hill climbing on the Greedy Randomized initial solution
    [sol, ~]= hillClimbing(sP, nSP, T, nNodes, Links, sol, energy, L, alpha,
C);
    Loads= calculateLinkLoads(nNodes,Links,T,sP,sol);
    energy= calculateEnergyConsumption(L, Loads, T, sol, sP);

    % Update the best solution if a better one is found
    if energy < bestEnergy
        bestSol = sol;
        bestEnergy= energy;
        bestLoads = Loads;
        bestLoadTime= toc(t);
    end
(...)
```

The same adjustments discussed above were also carried out in the `hill_climbing` function.

Exercise 2.b

Insertions or Changes in the code

As necessary in task 1, the obtained optimized solution is shown to the user, upon the calculation of various parameters. This function can be seen below:

```
function print_solution_stats(Loads, energy, contador, bestLoadTime, sol,
sP, T1, T2, D)
    fprintf("Solution stats: \n");

    maxLoad= max(max(Loads(:,3:4)));
    fprintf(" - Worst link load: %.2f Gbps\n", maxLoad);

    averageLinkLoad= mean(Loads(:,3:4));
    fprintf(" - Average Upload link load: %.2f Gbps\n", averageLinkLoad(1));
    fprintf(" - Average Download link load: %.2f Gbps\n",
averageLinkLoad(2));

    fprintf(" - Network energy consumption: %.2f\n", energy);

    % Value per service
    service1Count= 0;
    service2Count= 0;
    for i=1:length(sol)
        % get path of the solution
        path= sP{i}{sol(i)};

        % calculate path delay
        delay= 0;
        for ii=1:(length(path)-1)
            delay = delay + D(path(ii), path(ii+1));
        end

        if i <= length(T1) % service 1
            service1Count= service1Count + delay;
        else % service 2
            service2Count= service2Count + delay;
        end
    end

    fprintf(" - Average Round trip time:\n");
    service1RTT= service1Count/length(T1);
    fprintf(" \t- Service 1: %.3f ms\n", service1RTT * 2 * 1000);
    service2RTT= service2Count/length(T2);
    fprintf(" \t- Service 2: %.3f ms\n", service2RTT * 2 * 1000);
    fprintf(" - N° of links without traffic: %d\n", max(sum(Loads(:,
3:4)==0)));
    fprintf(' - Links with no traffic: ');
    for i = 1:length(Loads)
        if sum(Loads(i, 3:4)) == 0
            fprintf('{%d,%d} ', Loads(i,1), Loads(i,2));
        end
    end
    fprintf('\n');

    fprintf(" - Number of cycles: %d\n", contador);

    fprintf(" - Best solution time: %.2f\n", bestLoadTime);

    fprintf("\n");

end
```

Having updated the Multi-Start algorithm to prioritize lower energy costs and completed the print solution function, the remaining task involves computing the initial solutions for *k*=2 and integrating the previous functions. It's important to note that in optimization, both services 1 and 2 are calculated together as one set of flows. However, when displaying the solution, we separate these services to calculate their respective **Average Round-Trip Time**. Here's the main code for this exercise:

```
load('InputDataProject2.mat')

nNodes= size(Nodes,1);
nLinks= size(Links,1);
T= [T1; T2];
nFlows= size(T,1);
v= 2*10^5;
D= L/v;

fprintf('Exercise 2.b:\n');

% Computing up to k=2 link disjoint paths
%   for all flows from 1 to nFlows:
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
% sP{f}{i} is the i-th path of flow f
% nSP(f) is the number of paths of flow f

runtimeLimint= 60;
alpha= 1; % support max link load
```

```
C= 100; % Link capacity per direction
[sol, Loads, energy, contador, ~, bestLoadTime] =
multiStartHillClimbingGreedy(sP, nSP, T, nNodes, Links, runtimeLimint,
alpha, L, C);

print_solution_stats(Loads, energy, contador, bestLoadTime, sol, sP, T1, T2,
D);
```

Results

Executing our algorithm for 60 seconds with *k=2* resulted in the following output:

```
Exercise 2.b:
Solution stats:
 - Worst link load: 99.80 Gbps
 - Average Upload link load: 38.22 Gbps
 - Average Download link load: 38.03 Gbps
 - Network energy consumption: 2375.44
 - Average Round trip time:
        - Service 1: 5.384 ms
        - Service 2: 5.751 ms
 - N° of links without traffic: 8
 - Links with no traffic: {1,2} {1,7} {2,3} {3,6} {3,8} {6,15} {12,13}
{13,15}
 - Number of cycles: 4593
 - Best solution time: 0.24
```

Exercise 2.c

Executing our algorithm for 60 seconds with *k=6* resulted in the following output:

```
Exercise 2.c:
Solution stats:
 - Worst link load: 99.80 Gbps
 - Average Upload link load: 37.42 Gbps
 - Average Download link load: 37.28 Gbps
 - Network energy consumption: 2071.78
 - Average Round trip time:
        - Service 1: 5.567 ms
        - Service 2: 6.396 ms
 - N° of links without traffic: 13
 - Links with no traffic: {1,5} {1,7} {2,3} {2,5} {3,6} {3,8} {4,9} {6,14}
{11,13} {12,13} {12,14} {13,14} {14,15}
 - Number of cycles: 3594
 - Best solution time: 11.21
```

Exercise 2.d

Comparing both solutions, we can see that they maintain a consistent **Worst Link Load (WLL)** of *99.80 Gbps*, whilst having different solutions. This reflects that the increase of considered paths, lead the algorithm to find a more optimized solution.

Analyzing average link load, both *upload* and *download* showcase a small difference in load distribution across the network. While closely aligned, Exercise *2.c* manifests slightly lower average loads compared to Exercise *2.b*, indicating a that the obtained solution may marginally ease the burden on individual links.

In terms of energy consumption, we can see a substantial contrast between the two exercises, with Exercise *2.c* demonstrating a considerable reduction in overall energy utilization, registering only **2071.78**, comparing to **2375.44** from Exercise *2.d*. This different is reflected by the increase of links in sleeping mode with the solution of Exercise *2.c*.

The analysis of **round-trip times** for services reveals that Exercise *2.c* exhibits worse latency for both services compared to Exercise *2.b*. This is likely attributed to the obtained solution employing longer links in favor of reusing links already carrying traffic, leading to a slight increase in overall latency.

As referred before, using *k=6* instead of *k=2* increased the possible paths taken by each node and thus enabled the algorithm to locate a better solution, with lower energy consumption and more links in sleep (**13** compared to **8**).

When it comes to the number of cycles executed by the different exercises. The differences can be justified by the increased computations with handling of more paths in Exercise *2.c*, compared to Exercise *2.b*.

In terms of performance. While Exercise *2.b* identified it's optimal solution within 240 ms of execution. Exercise *2.c* took considerably more time, with the optimal solution only being discovered after 11.21 seconds. This showed that the difference the solution obtained by exercise *2.c* is much specific. But the biggest drawn from this is that the probability of executing the algorithm for more than 12 seconds and finding a better solution is very slim. Indicating that there is no need to run the algorithm for more than the specified 60 seconds.

In summary, while both solutions are similar in certain aspects like **Worst Link Load (WLL)**, Exercise *2.c* stands out for it's reduced energy consumption and network utilization. Nonetheless, this optimization comes at the expense of slightly higher round-trip delay, especially noticeable in Service 2, and increased computational effort.

Exercise 2.e

While both exercise *1.d* and *2.c* maintain a focus on optimizing network performance, Exercise *2.c* stands out for it's superior optimization in several key aspects compared to Exercise *1.c*.

Regarding the **Worst Link Load (WLL)**, Exercise *2.c* exhibits a higher **WLL** of 99.80 Gbps, while Exercise *1.c* showcases a lower **WLL** of 76.40 Gbps. This variance indicates that Exercise *2.c* is trying to stay below the limit of 100 Gbps for each flow direction and Exercise *1.c* is trying to optimize a solution for lower **WLL**.

Analyzing the average upload and download link loads, both exercises exhibit relatively close values, with Exercise *2.c* demonstrating slightly lower loads, indicating a more balanced load distribution across the network compared to Exercise *1.c*.

In terms of energy consumption, Exercise *2.c* reduces overall energy utilization to 2071.78, while Exercise *1.c* registers higher energy consumption at 2931.56. The significant discrepancy indicates that Exercise *2.c* is being able to reduce energy consumption by utilizing less links.

Examining round-trip times for services, Exercise *2.c*, manages to achieve better overall latency compared to Exercise *1.c*. This might be attributed to Exercise *1.d* use of more links, leading to a the utilization of longer links with higher delays.

Regarding the computational aspects, Exercise *2.c* takes notably longer to discover its optimal solution, requiring 11.21 seconds compared to Exercise *1.d* 3.93 seconds. This variance coupled with more than 3 times the number of cycles ran by Exercise *1.d*, compared to Exercise *2.c*, indicates that Exercise *1.d* is able to compute more paths and reach it's optimal solution much faster. Since both algorithms found it's solution within 12 seconds of execution, signals that there is no need to run the algorithm for more seconds in hopes of finding a better solution.

In summary, Exercise *2.c* demonstrates enhanced network optimization, especially in terms of reduced energy consumption and more balanced link utilization. However, this optimization does come at the expense of slightly higher round-trip delays and increased computational effort compared to Exercise *1.c*. Despite the higher **Worst Link Load**, Exercise *2.c* achieves an overall more optimized network solution, showcasing that an optimization based in energy consumption is a viable method to manage a network.

Exercise 3.a

`multiStartHillClimbingGreedy2` Function

This function performs Multi-Start Hill Climbing with Greedy Randomized Initialization to optimize a symmetrical single path routing solution for both services. The objective is to minimize the average round-trip propagation delay of the service with the worst average round-trip delay and then the average round-trip propagation delay of the other service.

```
function [bestSol, bestLoads, bestLoad, contador, somador, bestLoadTime] =
multiStartHillClimbingGreedy2(sP, nSP, T, nNodes, Links, D, timeLimit)
    % Initialize variables
    nFlows = size(T, 1);
    t= tic;
    bestLoad = inf;  % objective function value
    contador = 0;
    somador = 0;

    % Main loop with a time limit
    while toc(t) < timeLimit
        % Generate initial solution using Greedy Randomized approach
        sol = zeros(1, nFlows);
        for f = randperm(nFlows)
            auxBest = inf;
            for i = 1:nSP(f)
                sol(f) = i;
                auxLoads = calculateLinkLoads(nNodes, Links, T, sP, sol);
                auxLoad = max(max(auxLoads(:, 3:4)));
                if auxLoad < auxBest
                    auxBest = auxLoad;
                    ibest = i;
                end
            end
            sol(f) = ibest;
        end
        Loads = calculateLinkLoads(nNodes, Links, T, sP, sol);
        load = max(max(Loads(:, 3:4)));    % calculate max capacity of link
loads
```

```
        % Perform hill climbing on the Greedy Randomized initial solution
        [sol, load] = hillClimbing2(sP, nSP, T, nNodes, Links, sol, D);

        % Update the best solution if a better one is found
        if load < bestLoad
            bestSol = sol;
            bestLoad = load;
            bestLoads = Loads;
            bestLoadTime = toc(t);
        end

        contador = contador + 1;
        somador = somador + load;
    end
end
```

`hillClimbing2` Function

This function performs the hill climbing optimization on the Greedy Randomized initial solution. It explores different paths for each flow and evaluates the round-trip delays, considering weighted sums for the two services. The algorithm iteratively improves the solution until no further enhancement is possible.

```
function [sol, load] = hillClimbing2(sP, nSP, T, nNodes, Links, sol, D)
    nFlows = size(T, 1);

    % Identify the indices corresponding to T1 and T2 flows
    T1_indices = 1:size(T, 1);
    T2_indices = size(T, 1)+1:nFlows;

    bestLocalDelay = calculateRoundTripDelay(nNodes, Links, T, sP, sol, D);
    bestLocalSol = sol;
    improved = true;

    while improved
        improved = false;

        % Cycle through all flows
        for flow = 1:nFlows
            % Test each path
            for path = 1:nSP(flow)
                % Check that the current path is different from the one in
the solution
                if path ~= sol(flow)
                    % Create a new changed solution
                    auxSol = sol;
                    auxSol(flow) = path;

                    % Calculate the round-trip delay of this solution
                    auxDelay = calculateRoundTripDelay(nNodes, Links, T, sP,
auxSol, D);
```

```
                    % Weighted sum of round-trip delays, giving higher
weight to T1
                    weightedSumAuxDelay = sum(auxDelay(T1_indices)) + 0.5 *
sum(auxDelay(T2_indices));

                    % Weighted sum of round-trip delays for the best
solution found so far
                    weightedSumBestDelay = sum(bestLocalDelay(T1_indices)) +
0.5 * sum(bestLocalDelay(T2_indices));

                    % Check if this solution is better than the previous
found one
                    if weightedSumAuxDelay < weightedSumBestDelay
                        bestLocalDelay = auxDelay;
                        bestLocalSol = auxSol;
                    end
                end
            end
        end

        % If a better round-trip delay is found for a solution, change to it
        if max(bestLocalDelay(T1_indices)) <
max(calculateRoundTripDelay(nNodes, Links, T, sP, sol, D))
            sol = bestLocalSol;
            load = max(bestLocalDelay(T1_indices));
            improved = true;
        else
            % If no improvement is found, set load to the current value
            load = max(calculateRoundTripDelay(nNodes, Links, T, sP, sol,
D));
        end
    end
end
```

`calculateRoundTripDelay` Function

This function calculates the round-trip delay for each flow in the solution. It traverses the paths of each flow and computes the sum of propagation delays over the links.

```
function roundTripDelay = calculateRoundTripDelay(nNodes, Links, T, sP,
Solution, D)
    nFlows = size(T, 1);
    roundTripDelay = zeros(nFlows, 1);

    for i = 1:nFlows
        if Solution(i) > 0
            path = sP{i}{Solution(i)};
            for j = 2:length(path)
                roundTripDelay(i) = roundTripDelay(i) + 2 * D(path(j-1),
path(j));
            end
        end
        end
    end
```

This set of functions provides a modified algorithm to address the optimization problem specified in exercise 3.a, focusing on minimizing the average round-trip propagation delays for both services.

Exercise 3.b

For the algorithm run with k = 2:

Solution stats:

- **Worst link load:** 98.20 Gbps
- **Average Upload link load:** 37.94 Gbps
- **Average Download link load:** 38.21 Gbps
- **Network energy consumption:** 2922.68
- **Average Round trip time:**
    - Service 1: 5.432 ms
    - Service 2: 5.611 ms
- **Nº of links without traffic:** 3
- **Links with no traffic:** {1,7} {6,15} {12,13}
- **Number of cycles:** 75858
- **Best solution time:** 0.02

Exercise 3.c

For the algorithm run with k = 6:

Solution stats:

- **Worst link load:** 85.20 Gbps
- **Average Upload link load:** 36.41 Gbps
- **Average Download link load:** 36.11 Gbps
- **Network energy consumption:** 2876.71
- **Average Round trip time:**
    - Service 1: 5.642 ms

  - Service 2: 5.839 ms
- **Nº of links without traffic:** 4
- **Links with no traffic:** {3,8} {12,13} {12,14} {14,15}
- **Number of cycles:** 26247
- **Best solution time:** 0.04

Exercise 3.d

Comparison between k=2 and k=6:

The results indicate that increasing the value of k from 2 to 6 in the algorithm has led to improvements in various aspects:

- **Worst link load:** The worst link load reduced from 98.20 Gbps to 85.20 Gbps, indicating a more balanced distribution of traffic.
- **Average link loads:** Both average upload and download link loads decreased, demonstrating a more efficient utilization of network resources.
- **Network energy consumption:** The energy consumption also decreased, from 2922.68 to 2876.71, showcasing a more energy-efficient solution.
- **Average Round trip time:** Slight improvements were observed in the average round-trip times for both Service 1 and Service 2.
- **Number of links without traffic:** The number of links without traffic increased from 3 to 4, indicating a better overall utilization of links.

In summary, increasing the value of k has resulted in a solution with lower link loads, improved energy efficiency, and slightly reduced round-trip times.

Exercise 3.e

Metric	Exercise 1.d	Exercise 2.c	Exercise 3.c
Worst link load	76.40 Gbps	99.80 Gbps	85.20 Gbps
Avg. Upload link load	40.84 Gbps	40.10 Gbps	36.41 Gbps
Avg. Download link load	40.63 Gbps	40.39 Gbps	36.11 Gbps
Network energy consumption	2931.56	2071.34	2876.71
Avg. Round-trip time (Service 1)	6.882 ms	5.567 ms	5.642 ms

Metric	Exercise 1.d	Exercise 2.c	Exercise 3.c
Avg. Round-trip time (Service 2)	6.010 ms	6.591 ms	5.839 ms
Links without traffic	3	13	4
Cycles	12108	3531	26247
Best solution time	3.93 seconds	5.88 seconds	0.04 seconds

Comparison between Exercise 1.d, 2.c, and 3.c:

1. **Worst Link Load:** Exercise 1.d achieved the lowest worst link load (76.40 Gbps), followed by Exercise 3.c (85.20 Gbps) and Exercise 2.c (99.80 Gbps). This indicates that the optimization criteria differ, with Exercise 1.d focusing on minimizing link load, while Exercise 2.c prioritizes energy consumption.

2. **Average Link Load:** Exercise 1.d has higher average upload and download link loads compared to Exercise 3.c and Exercise 2.c, suggesting a more balanced distribution of traffic in the latter cases.

3. **Network Energy Consumption:** Exercise 1.d resulted in higher energy consumption (2931.56) compared to Exercise 3.c (2876.71) and Exercise 2.c (2071.34). This indicates a trade-off between link load and energy consumption.

4. **Round-trip Time:** Exercise 1.d resulted in higher round-trip times for both services compared to Exercise 3.c and Exercise 2.c. This suggests that the optimization criteria in Exercise 1.d may lead to higher delays.

5. **Number of Links Without Traffic:** Exercise 1.d had fewer links without traffic (3) compared to Exercise 3.c (4) and Exercise 2.c (13). This indicates better resource utilization in Exercise 1.d.

6. **Number of Cycles:** Exercise 1.d required more cycles (12108) compared to Exercise 3.c (26247) and Exercise 2.c (3531). This suggests that the algorithm in Exercise 1.d may take longer to converge.

7. **Best Solution Time:** Exercise 1.d achieved its best solution in a shorter time (3.93 seconds) compared to Exercise 3.c (0.04 seconds) and Exercise 2.c (5.88 seconds), indicating the efficiency of the algorithm in Exercise 1.d.

Conclusion

Exercise 1.d achieved the lowest worst link load, fewer links without traffic, and a competitive average link load. However, it had higher energy consumption and round-trip times compared to Exercise 2.c and Exercise 3.c. Exercise 3.c showed better performance in terms of worst link load, average link load, and energy consumption, with faster convergence (lower cycles) compared to Exercise 2.c. The choice between the three solutions depends on the specific priorities of the network operator, whether it be minimizing link load, energy consumption, or achieving a balance between different metrics.

Exercise 4.a

The anycast Source Nodes present in `T3` matrix don't have the closest anycast node already defined, so we created the `anycastDestination` function to determine it from a matrix of nodes. In this case, the options are **node 3** and **node 10**. This function will compute `k` paths for the combination of source and destination nodes, and then determine for each source node, the closest (lowest round-trip propagation delay) anycast node, saving it's generated paths. This function can be found below:

```
function [costs, sP, nSP]= anycastDestination(nodes, anycastNodes, D, k)
    nNodes= length(nodes);
    sP= cell(1, nNodes);
    nSP= zeros(1,nNodes);
    costs= zeros(k, nNodes);

    for n= 1:nNodes
        best= inf;

        if ismember(nodes(n), anycastNodes)
            costs(n)= 0;    % anycast node has zero cost to itself
        else
            % Choose the anycast node with the smallest delay
            for a = 1:length(anycastNodes)
                [shortestPath, totalCost]= kShortestPath(D, nodes(n),
anycastNodes(a), k);

                % Check if the current total cost is better than the best
cost found so far
                if totalCost < best
                    sP{n}= shortestPath;
                    nSP(n)= length(totalCost);
                    best= totalCost;
                    costs(:,n)= totalCost;
                end
            end
        end
    end
end
```

Since another service was added, the `print_solution_stats` was also altered to display the round-trip propagation delay of the new service. The modifications can be seen below:

```
(...)
    if i <= length(T1) % service 1
        service1Count= service1Count + delay;
    elseif i <= (length(T1) + length(T2)) % service 2
        service2Count= service2Count + delay;
    else % service 3
        service3Count= service3Count + delay;
    end
end

(...)
service3RTT= service3Count/length(T3);
fprintf(" \t- Service 3: %.3f ms\n", service3RTT * 2 * 1000);
```

Exercise 4.b

Modification or Insertions in the code

The main code can be found below. Essentially, it starts by computing the `k` unicast flows and then compute the destination flow of each anycast flow. Next, we compute the duplicate `T3` matrix, which has a second column corresponding to the closes anycast node and concatenate this new matrix with the pre-existing `T` matrix. In the last part, we just run the optimization algorithm and print the solution.

```
(...)

fprintf('Exercise 4.b:\n');

% Computing up to k=6 link disjoint paths for all unicast flows:
k= 6;
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

% Computing up to k=6 link disjoint paths for all anycast flows
sourceAnycastNodes= T3(:,1);
[costs, tempsP, tempnSP]= anycastDestination(sourceAnycastNodes,
anycastNodes, D, k);

% Integrate anycast flows paths and counts into sP and nSP, after unicast
dupT3= zeros(size(T3));
for i = 1:length(tempsP)
    sP{nFlows + i} = tempsP{i};
    nSP(nFlows + i) = tempnSP(i);
    % Get destination anycast node
    tempPath= tempsP{i}{1};
    dest= tempPath(length(tempPath));
    % change T3
    dupT3(i,1)= T3(i,1);
    dupT3(i,2)= dest;
    dupT3(i,3)= T3(i,2);
    dupT3(i,4)= T3(i,3);
end

% join unicast and anycast flows
T= [T; dupT3];
nFlows= size(T,1);

runtimeLimint= 60;
alpha= 1; % support max link load
C= 100; % Link capacity per direction
[sol, Loads, energy, contador, ~, bestLoadTime] =
multiStartHillClimbingGreedy(sP, nSP, T, nNodes, Links, runtimeLimint,
alpha, L, C);

print_solution_stats(Loads, energy, contador, bestLoadTime, sol, sP, T1, T2,
dupT3, D);
```

Results

Executing our algorithm generated the following results:

```
Exercise 4.b:
Solution stats:
 - Worst link load: 99.30 Gbps
 - Average Upload link load: 45.09 Gbps
 - Average Download link load: 42.60 Gbps
 - Network energy consumption: 2337.10
 - Average Round trip time:
        - Service 1: 5.623 ms
        - Service 2: 6.139 ms
        - Service 3: 6.187 ms
 - N° of links without traffic: 9
 - Links without traffic: {1,2} {1,7} {3,8} {6,8} {6,15} {9,10} {11,13}
```

```
{13,14} {13,15}
 - Number of cycles: 1533
 - Best solution time: 27.30
```

Exercise 4.c

Modification or Insertions in the code

Based on Exercise *4.b*, we start by making combinations with all nodes and then test each combination, to select the anycast node pair with lowest cost. Worth mentioning that combinations where one of the nodes will also be an anycast source node, will have an advantage, since one of the flows will have a cost equal to `0`. The code can be observed below:

```
fprintf('\nExercise 4.c:\n');
combsAnycastSources= nchoosek(1:nNodes, 2);

% Initialization
k= 6;
bestWorstCost= inf;
bestAnycastNodes= combsAnycastSources(1, :); % initialize with the first
pair
bestSP= cell(1, length(sourceAnycastNodes));
bestnSP= zeros(1, size(T3,1));

% Loop through combinations
for i = 1:size(combsAnycastSources,1)
    anycastNodes= combsAnycastSources(i, :);

    % Test combination of nodes
    [costs, tempsP, tempnSP]= anycastDestination(sourceAnycastNodes,
anycastNodes, D, k);
    cost = sum(sum(costs));

    % Update best values if better result is found
    % and exclude anycast destination nodes originating in source nodes
    if cost < bestWorstCost % && (min(min(costs)) > 0)
        bestWorstCost= cost;
        bestAnycastNodes= anycastNodes;
        bestSP= tempsP;
        bestnSP= tempnSP;
    end
end

fprintf("Best Node combination: {%d, %d}\n", bestAnycastNodes);
```

Results

Executing this code, resulted in the following text:

```
Exercise 4.c:
Best Node combination: {4, 13}
```

Exercise 4.d

Modification or Insertions in the code

Adapting the code from Exercise *4.b*, we introduce the ability to remove flows from the `T3` duplicate matrix, when the source and destination anycast flow is the same and then pad the difference with `0`, when the optimization algorithm is ran. This is done to be able to accurately process the Round Trip time, since we use the length of `T3` matrix inside of the `print_solution_stats` function, so it needs to have all flows. However, while calculating a solution, the `T` matrix can't have flows originating and terminating in the same flow.

```
(...)
% Integrate anycast flows paths and counts into sP and nSP, after unicast
dupT3= zeros(1);
aux= 1;
for i = 1:length(bestSP)
    if ~isempty(bestSP{i})
        sP{nFlows + aux} = bestSP{i};
        nSP(nFlows + aux) = bestnSP(i);

        % Get destination anycast node
        tempPath= bestSP{i}{1};
        dest= tempPath(length(tempPath));

        % change T3
        dupT3(aux,1)= T3(i,1);
        dupT3(aux,2)= dest;
        dupT3(aux,3)= T3(i,2);
        dupT3(aux,4)= T3(i,3);

        aux = aux + 1;
    end
end

(...)
```

```
[sol, Loads, energy, contador, ~, bestLoadTime] =
multiStartHillClimbingGreedy(sP, nSP, T, nNodes, Links, runtimeLimint,
alpha, L, C);

% add the flows removed before, that have the same source and destination
% anycast node
numRowsT3 = size(T3, 1);
numRowsdupT3 = size(dupT3, 1);

if numRowsdupT3 < numRowsT3
    rowsToAdd = numRowsT3 - numRowsdupT3;

    % Append rows of zeros to matrix dupT3
    dupT3 = [dupT3; zeros(rowsToAdd, size(dupT3, 2))];
end

print_solution_stats(Loads, energy, contador, bestLoadTime, sol, sP, T1, T2,
dupT3, D);
```

Results

```
Exercise 4.d:
Solution stats:
 - Worst link load: 99.80 Gbps
 - Average Upload link load: 42.41 Gbps
 - Average Download link load: 42.12 Gbps
 - Network energy consumption: 2210.23
 - Average Round trip time:
        - Service 1: 5.702 ms
        - Service 2: 5.930 ms
        - Service 3: 3.963 ms
 - N° of links without traffic: 11
 - Links without traffic: {1,5} {1,7} {2,3} {2,5} {3,6} {3,8} {4,10} {6,15}
{11,13} {13,14} {13,15}
 - Number of cycles: 1886
 - Best solution time: 34.30
```

Exercise 4.e

In comparing the solutions derived from Exercises *4.b* and *4.d*, notable differences surface across multiple performance metrics, shedding light on the nuances between these network optimization approaches.

Firstly, considering the **Worst Link Load**, Exercise *4.d* marginally surpasses Exercise *4.b*, registering a slightly higher load of 99.80 Gbps compared to 99.20 Gbps in Exercise *4.b*. This isn't necessarily something bad, since it's within the link load limit of 100 Gbps so it can be attributed just to the difference in the solution.

Examining the **Average Upload and Download Link Loads**, Exercise *4.b* demonstrates higher average loads with an upload rate of 46.20 Gbps and a download rate of 43.02 Gbps, whereas Exercise *4.d* presents lower figures with an upload load of 42.41 Gbps and a download load of 42.12 Gbps. This suggests a more balanced distribution of loads in Exercise *4.d*, with a solution that poses overall less load on the system, due to it not having as much redundant links.

In terms of **Network Energy Consumption**, Exercise *4.d* significantly outperforms Exercise *4.b*, showcasing a lower energy consumption rate of 2210.23 compared to 2340.27. This is because of the solution achieved in Exercise *4.d*, which has more links in sleep or less **Links without Traffic**, thus consuming less energy.

Analyzing the **Average Round Trip Times** for services, Exercise *4.d* displays general improvements compared to Exercise *4.b*. The improvements in the unicast flows are almost negligible and can be dismissed as just a small difference in the solution. However, the anycast flow suffered a major improvement, due to the calculation of the optimal pair of anycast destination nodes made in Exercise *4.c*.

In terms of **Best Solution Time**, both exercises achieved their solution in respectable times, taking no more than 35 seconds to achieve the optimal solution, signalling that there isn't a need to run the algorithm for more than the current 60 seconds.

In conclusion, Exercise *4.d* yields a more efficient network solution, marked by reduced energy consumption, better-balanced loads, improved latency for services, and even being able to analyze more solutions than Exercise *4.b*. This results reflect the need to simulate and choose the optimal anycast service location before installing it in any node.