TMA521 – Large scale optimization, 11th February 2013

VLSI routing and Lagrangian duality

Simon Sigurdhsson, ⟨ssimon@student.chalmers.se⟩

Abstract This report discusses a solution to the first project of the Large scale optimization course (TMA521) given by the Applied mathematics department at Chalmers University of Technology. The problem considered is a routing problem in VLSI based on that discussed by Feo and Hochbaum (1986).

1 Introduction

Although the problem is defined and discussed by both Feo and Hochbaum (1986) and the project description, it will now be briefly presented to provide some context.

The problem consists of deciding wether it is possible to connect a number of components in the context of a two-layer board with horizontal wiring on one side and vertical on the other, and a specified number of connectors between these layers.

The problem is modelled mathematically as an ILP problem which is then Lagrangian relaxed resulting essentialy in one cheapest route problem for each wanted connection.

2 Subgradient optimization

The first task of the project consists of implementing the subgradient optimization as MATLAB code. Routines that solve the lagrangian subproblems (the cheapest route problems) are given, and as such the code only has to reinterpret the output of those routines and perform the subgradient optimization.

Appendices A on page 7 shows the complete code listing for this task (sans the parts given in the problem description, *i.e.* `gsp.c`, `sph.c`, `visagrid.m` and the problem instance files), but a full description of the algorithm is given below.

2.1 The subgradient algorithm

Since the given function `gsp.c` solves the actual lagrangian subproblems, the implementation of the subgradient algorithm is very simple:

1. Call `gsp` to solve the subproblem, discard all paths that have a total cost of 1 or more, and transform the data into an x_{ijl} matrix. This is what the functions `okcom` and `getxij` (described in appendices A.1 to A.2 on page 10) do, respectively.
2. Calculate the dual value for the solution obtained in the current iteration,

$$h(\pi^t) = \sum_{i=1}^{n} \pi_i^t + \sum_{l=1}^{k} \left(x_{t_l s_l l} - \sum_{i=1}^{n} \sum_{j=1}^{n} \pi_i^t x_{jil} \right).$$

3. Calculate the subgradient direction,

$$d_i^t = 1 - \sum_{l=1}^{k} \sum_{j=1}^{n} x_{jil}.$$

4. Calculate the step length,

$$s^t = \lambda^t \frac{h(\pi) - LBD}{\sum_{i=1}^{n} \left(d_i^t \right)^2}.$$

5. Update the dual variables by taking a step in the subgradient direction, *i.e.* set $\pi_i^{t+1} = \max\{0, \pi_i^t - s^t d_i^t\}$.
6. Finally, decrease lambda by setting $\lambda^{t+1} = 0.95\lambda^t$.
7. Repeat from step 1 unless the maximum number of iteration has been reached.

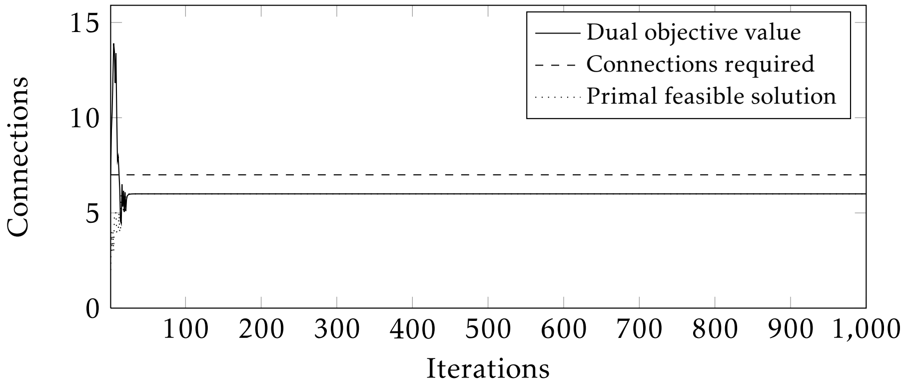
2.2 Results

As shown by figure 1 on the following page, the only problem instance that can be classified as either possible or impossible without using any primal feasibility heuristic is instance p6 (figure 1a on the next page), which has a dual objective value of approximately 6, meaning the optimal value of the problem (*i.e.* the maximum number of connections possible) is at most 6. Thus, the required number of connections (7) cannot be obtained.

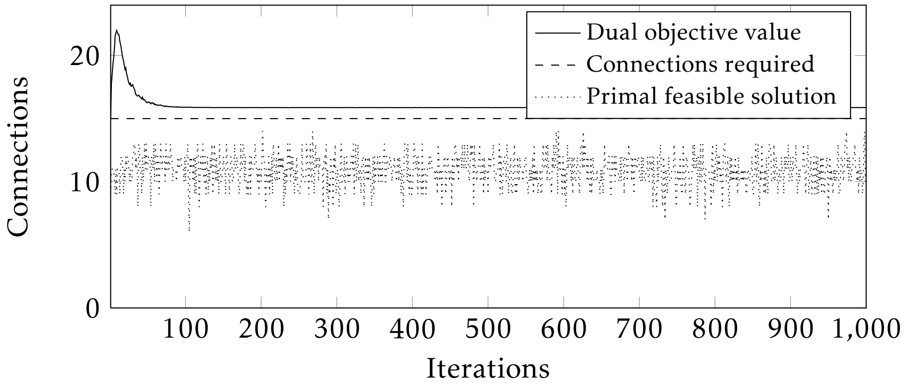
The other two problems show dual objective values larger than the required number of connections, but this is no guarantee that the instance can have that many connections; the dual objective value it an optimistic bound.

3 A feasibility heuristic

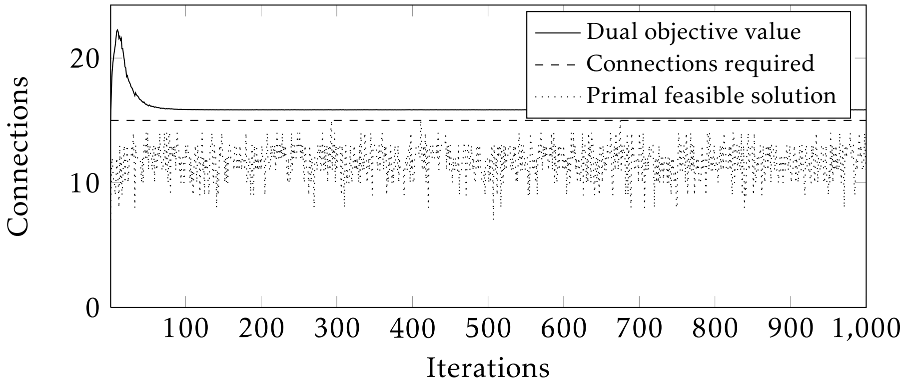
Implementing a heuristic that uses the problem formulation to reduce a dual solution to a feasible one might seem like a daunting task. However, thinking of the



(a)*Instance p6.*



(b)*Instance p10.*



(c)*Instance p11.*

Figure 1: *The dual objective value and primal feasible solution for three problem instances.*

subproblems as cheapest route problems, they only differ from feasible solutions of the original problem in that the paths are not required to be vertex-disjoint. Thus, removing or re-routing paths that cross in a dual solution will result in a feasible solution to the original problem, even though it may be a bad one that doesn't connect all pairs.

The heuristic implemented in this task uses that very idea. Starting with a dual solution, knowing that it is a collection of paths, the heuristic calculates the number of paths passing each node of the problem (*i.e.* $n_i = \sum_{l=1}^{k} z_{il}$, extending the notation used on page 4 of the project description). The algorithm then proceeds as follows:

1. Calculate n_i.
2. If $n_i < 2, \forall i$, abort the heuristic (since we evidently have a feasible solution).
3. Select a path p passing through any node with $n_i \geq 2$.
4. Let p_s and p_e be the first and last nodes of p, respectively. Remove p from the dual solution.
5. Find a cheapest path p' from p_s to p_e, with costs

$$c_i = \begin{cases} \pi_i, & \text{no path through node } i \\ \infty, & \text{otherwise.} \end{cases}$$

6. If $\text{cost}(p') < \infty$, add p' to the dual solution.
7. Repeat from step 1.

The algorithm will always terminate since it keeps removing paths from the overpopulated nodes either by finding an alternative route or by simply discarding the path. As such, the heuristic may (although this should be rare) terminate with a (still feasible) solution containing no paths at all. Generally it should be able to return at least one path, however.

The algorithm has a worst-case complexity $O(kn')$, where n' is the number of overpopulated nodes of the dual solution. Since the number of overpopulated nodes is less than the total number of nodes in the problem, we can rewrite the complexity as $O(k|\mathcal{V}|) = O(kn)$. The number of connections must also be less than $n/2$ due to the problem structure, and as such the complexity is $O(n^2)$, which is perfectly reasonable for a feasibility heuristic.

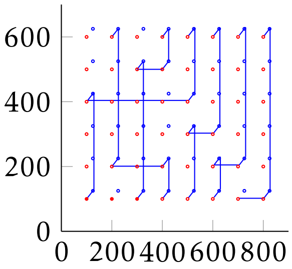
3.1 Results

As shown by figure 1 on the preceding page, the heuristic performs well for instance p6, finding a feasible solution that matches the dual objective value quite early in the subgradient algorithm. Instance p10 seems more difficult, and in that case the heuristic fails to match the dual value and in fact provides no additional information (*i.e.* there is no way of telling if the actual optimum is above or below k). In instance `p11`, however, the heuristic finds a couple of feasible soluions which connect k pairs, which means we can conclude that the optimum value is at least k.

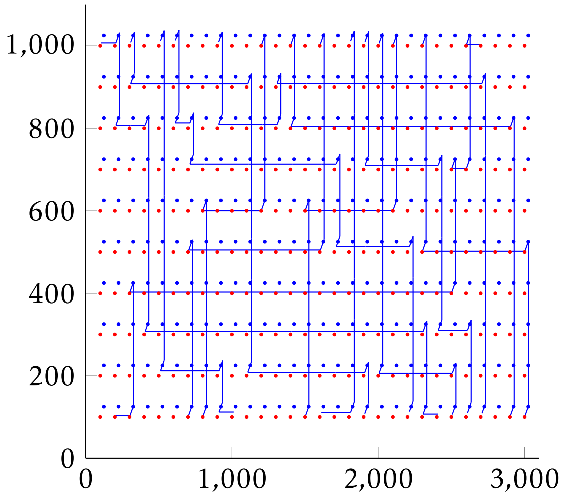
To summarize, the combination of Lagrangian relaxation, the subgradient algorithm and a feasibility heuristic indicates that `p6` has no solution to the wiring problem, `p11` has a solution to the wiring problem, and that `p10` may have a solution to the wiring problem. Figure 2 on the next page shows the best feasible solution produced by the heuristic for each of these instances, and one can verify that all these solutions are in fact feasible by checking for overused nodes, and one can also see that they have 6, 14 and 15 paths respectively (where k is 7, 15 and 15 for the three problem instances).

References

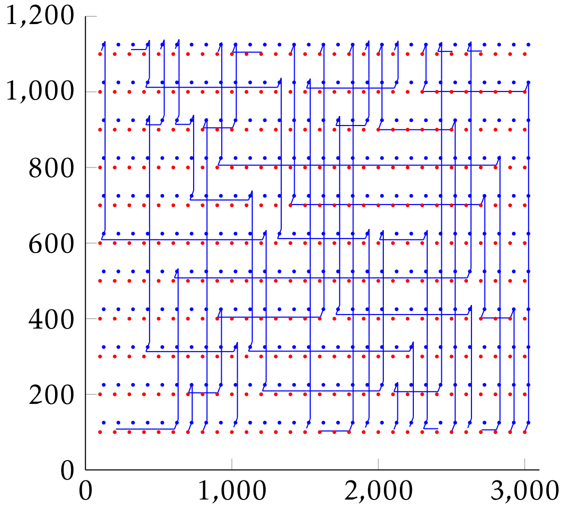
Feo, T. A. and D. S. Hochbaum (Nov. 1986). ‘Lagrangian Relaxation for Testing Infeasibility in VLSI Routing’. In: 34.6, pp. 819–831.



(a)*Instance p6.*



(b)*Instance p10.*



(c)*Instance p11.*

Figure 2: *The best primal feasible solution for three problem instances.*

A Program code

```
% TMA521 - Large scale optimization
% Spring 2013
% Project 1, tasks 1 & 2
% Simon Sigurdhsson

% This file is the main file that solves (after editing according to
% instructions in comments) tasks 1 and 2 of the project.

% Files have been organized such that given code (gsp.c, sph.c and
% visagrid.m) resides in a subfolder "given", and the instance files
% are in the "instances" subfolder. Hence, we need to add these to
% the PATH in order to use these files.
% Additionally, all variables are cleared and all figures closed at
% the start of the program, to avoid confusion from earlier results.
addpath('given','instances')
clear all; close all; clc;

% Initialization of variables
% This is where all non-local variables of the program are defined.
% First, we have a couple of variables that control the program flow,
% deciding what instance we are solving and how many iterations we
% should run the subgradient solver for.
p11;              % Problem to solve.
maxIter = 1000;   % Maximum number of subgradient iterations.
% Then, we initialize some of the variables used in the subgradient
% algorithm, so that we don't have to reallocate them inside the
% loop (this is very inefficient). All these variables are set to
% 0 initially, which makes for a good starting value for pi and will
% be overwritten for all other variables, except for the list of
% upper bounds, which is set to Inf (so that all calculated upper
% bounds are smaller), and lambda which is set to 2 in accordance
% with the project description.
pi = zeros(maxIter, dimX*dimY*2); % Lagrangian multipliers for each time step
d = zeros(maxIter, dimX*dimY*2);  % Subgradient direction for each time step
s = zeros(maxIter, 1);            % Step length for each time step
UBDS = ones(maxIter, 1)*Inf;      % Upper bound for each time step
LBDS = zeros(size(UBDS));         % Lower bound for each time step
lambda = 2;                       % Step length modifier
optCom = [];                      % Storage for best feasible solution (com)
optNl = [];                       % Storage for best feasible solution (nl)
optPi = [];                       % Storage for best feasible solution (pi)

% The subgradient scheme (solver)
% This is where the actual work begins. The only stopping criterion
% used is the maximum number of iterations, and the code inside the
% loop pretty much follows the flow chart given in the project
% description.
for t=1:maxIter
    % 1. Solve the Lagrangian subproblems
    % The Lagrangian subproblems are solved as cheapest-path problems
    % by the given function gsp. Since the output of gsp is hard to
    % handle, a function getxij (see getxij.m) is used to transform
```

```
    % the output to two 3-dimensional matrices describing the problem
    % variables x_{ijl} and x_{t_{l}s_{l}l}. But first, the okcom
    % function (see okcom.m) eliminates the paths with cost larger
    % than 1, as described in the project description.
    nl = gsp(dimX,dimY,pi(t,:)',k,com);           % Solve the subproblems
    [ok, oknl] = okcom(pi(t,:),k,com,nl);         % Which paths are "ok"?
    kok = find(ismember(com, ok, 'rows') == 1);   % Which rows of com are "ok"?
    [xij, xtlsl] = getxij(dimX,dimY,k,com,nl,kok);% Transform to x_{ijl} matrix

    % 1.1. Calculate the primal feasibility heuristic (only for task 2)
    % The heuristic, described in heuristic.m, creates a feasible
    % solution based on the current solution. It returns data of the
    % same structure as gsp, so it too requires getxij to transform
    % the output into similar matrices.
    [hcom, hnl] = heuristic(dimX, dimY, pi(t,:), k, com, nl);       % Find primal feasible so
    [hxij, hxtlsl] = getxij(dimX,dimY,k,hcom,hnl,(1:length(hcom))');% Transform to x_{ijl} ma

    % 2. Calculate upper bound, h(pi)
    % The upper bound, essentially the Lagrangian dual value, is
    % calculated as described in the project description. Since the
    % output of gsp has been transformed into a matrix, the actual
    % calculation is very similar to the mathematical formula shown
    % in the project description.
    lpi = repmat(pi(t,:)', [1 dimX*dimY*2 k]);                  % Expand pi, for element-wise
    h = sum(pi(t,:)) + sum(xtlsl - sum(sum(lpi.*xij, 1), 2), 3);% Calculate h(pi) according t

    % 2.1. Calculate the lower bound (only for task 2)
    % The lower bound is given by the primal feasible solution found
    % by the heuristic. Since it is a *primal* feasible solution, the
    % lower bound isn't calculated using the Lagrangian dual value but
    % using the original problem formulation. Thus, it is a bit simpler
    % than the calculation of h(pi), but still analogous to the math
    % described in the project description.
    LBDS(t) = sum(hxtlsl, 3); % Count the number of connections made by the primal feasible s
    % Note that since LBDS(t) = 0 for all t before this assignment is
    % made, simply commenting the above line (along with the code
    % under 1.1) will essentialy solve task 1 instead of task 2.

    % 3. Calculate subgradient direction d
    % Again, since the output of gsp has been transformed into a
    % suitable matrix, the calculation of the subgradient direction is
    % very similar to the mathematical formula given by the project
    % description.
    sxjl = sum(sum(xij,1),3);      % Pre-calculate sum (for efficiency)
    d(t,:) = ones(size(sxjl))-sxjl;% Calculate subgradient direction

    % 4. Calculate step length s
    % The step length calculation is also pretty much a verbatim
    % translation of the mathematical formula into MATLAB code.
    % Note that in task 2 we actually use the lower bound given by
    % the heuristic in the formula, while LBDS=0 for task 1 as
    % suggested by the project description.
    s(t) = lambda*(h-LBDS(t))/sum(d(t,:).^2);% Calculate step length

```

```
    % 5. Take step s in direction -d
    % If this isn't the last iteration, the Lagrangian multipliers
    % have to be updated for the next iteration by taking a step in
    % the subgradient direction. Again (there's a pattern here, no?),
    % the transformation of the gsp output makes this a verbatim
    % translation of math into MATLAB code.
    if t ~= maxIter                                               % If this isn't the last it
       pi(t+1,:) = max(zeros(size(pi(t,:))),pi(t,:)-s(t).*d(t,:));% Take a step in the subgra
    end

    % 6. Decrease value of lambda
    % In accordance with the project description, lambda is decreased
    % by 5% every iteration. This ensures 0<lambda<2.
    lambda = lambda*0.95;

    % 7. Save for plotting
    % In order to present graphs showing the convergence of the
    % subgradient algorithm along with the best solution found by
    % the heuristic (for task 2), the data is saved in the
    % preallocated variables.
    if (LBDS(t) >= max(LBDS)) % && false % Uncomment "&& false" for task 1
        optCom = hcom(any(hcom,2),:);    % Save the best feasible "com"
        optNl = hnl;                     % Save the best feasible "nl"
        optPi = pi;                      % Save the best feasible "pi"
    end
    UBDS(t) = h;                         % Save the upper bound of the current iteration

    % 8. Progress bar (sort of)
    % This simply prints some text every 25th iteration, to roughly
    % indicate how far the algorithm has come and how long we have
    % to wait before it will terminate.
    if mod(t,25) == 0
       disp(['Iteration␣', num2str(t), '␣of␣', num2str(maxIter)])
    end
end

% Plotting results
% The subgradient algorithm is done, and it is time to show the
% results. First, a plot showing the convergence of the subgradient
% algorithm and the heuristic (in task 2) is shown. Since all data
% has been saved, this is fairly standard and boring MATLAB code.
plot(1:maxIter, UBDS, 'k-'); hold on;       % Plot the upper bounds
plot(1:maxIter, k*ones(size(UBDS)), 'k--'); % Indicate the number of connections we want
plot(1:maxIter, LBDS, 'k:');                % Plot the lower bounds (only useful in task 2)
axis([1 maxIter 0 max(k, max(UBDS))+2]);    % Set sensible axes
xlabel('Iterations');                       % Label the x axis
ylabel('Connections');                      % Label the y axis
legend('Dual␣objective␣value','Connections␣required','Primal␣feasible␣solution'); % Explain p
% Next (only for task 2, comment out if running task 1), the best
% primal feasible solution found by the heuristic is shown using
% the given visagrid function. Again, nothing strange happening.
figure;                                     % Get a new figure
visagrid(dimX,dimY,optNl,optCom,optPi,25);  % Show the feasible solution
```

A.1 The `okcom` function

```
% TMA521 - Large scale optimization
% Spring 2013
% Project 1, tasks 1 & 2
% Simon Sigurdhsson

function [ ok, newnl ] = okcom( pi, k, com, nl )
%OKCOM Eliminates paths from com/nl if their cost is large.
%   The okcom function, which contains code from page 6 of
%   the project description, calculates the cost of each path
%   in com/nl, adding the path to ok/newnl if the cost is less
%   than one.
    last = 0;
    ok = zeros(k,2);
    newnl = [];
    for i = 1:k
        first = last+1;
        slask = find(nl(last+1:length(nl)) == com(i,1));
        last = slask(1)+first-1;
        if sum(pi(nl(first:last))) < 1
            ok(i,:) = com(i,:);
            newnl = [newnl; nl(first:last)];
        end
    end
end
```

A.2 The `getxij` function

```
% TMA521 - Large scale optimization
% Spring 2013
% Project 1, tasks 1 & 2
% Simon Sigurdhsson

function [ xij, xtlsl ] = getxij( dimX, dimY, k, com, nl, kok )
%GETXIJ Calculate x_{ijl} and x_{t_{l}s_{l}l} matrices
%   The getxij function takes the output of gsp and transforms it
%   into the actual problem variables x_{ijl} and x_{t_{l}s_{l}l},
%   making it much easier to calculate the dual value, subgradient
%   direction and step length required by the subgradient algorithm.
    % To begin with, a couple of local variables are defined.
    maxij = dimX*dimY*2;         % The number of nodes in the problem
    xij = zeros(maxij, maxij, k);% Output matrix, preallocated
    xtlsl = zeros(1, 1, k);      % Output matrix, preallocated
    tempnl = nl;                 % Copy of nl, will be modified in loop
    % Now, for all the "ok" paths in com/nl, we set the appropriate
    % elements of the output matrices to 1 (remember that they are
    % initialized to 0).
    for i=kok'
        % First, the part of nl containing the ith path is found.
        % It is assumed that the paths in nl have the same order as
        % the corresponding path endpoints in com, but that the path
        % is stored "backwards".
        % First, we extract the endpoints.
```

```
    sn = com(i,1); en = com(i,2);
    % Then, we extract the path corresponding to those endpoints,
    % assuming that the path is the next one in nl (i.e. the first
    % one in tempnl, in which we discard each path after finding it).
    thisnl = tempnl(1:find(tempnl == sn));
    % Discard the path we just found from tempnl.
    tempnl = tempnl((find(tempnl == sn)+1):end);
    % Now, we set all the appropriate elements of x_{ijl} (with l=i)
    % to one. Since the path is backwards and the matrix is used as
    % x_{jil},for each element j in thisnl except the last, we set
    % xij(j, j+1, i) to one. This is done using sub2ind, since just
    % inserting the vectors with ordinary subscript indexing sets
    % entire blocks of the matrix (which of course is incorrect).
    xij(sub2ind(size(xij),thisnl(1:end-1),thisnl(2:end),i*ones(length(thisnl)-1,1))) = 1;
    % Finally, we set x_{t_{l}s_{l}l} to one, since this path
    % obviously forms a connection.
    xtlsl(1,1,i) = 1;
   end
end
```

A.3 The primal feasibility `heuristic` function

```
% TMA521 - Large scale optimization
% Spring 2013
% Project 1, task 2
% Simon Sigurdhsson

% This is the primal feasibility heuristic written for task 2, and it
% is explained further in the report.

function [ ncom, nnl ] = heuristic( dimX, dimY, pi, ~, com, nl)
%HEURISTIC Finds a primal feasible solution given a Lagrangian dual solution
%   This heuristic, explained further by the report, basically transforms
%   a dual solution to a primal feasible solution by re-routing and/or
%   discarding paths from the dual solution. It is polynomial in the number
%   of nodes of the problem (in fact, O(n^2)), and will always terminate with
%   a feasible (but potentially very bad) solution.
  % First, we save the input variables since we'll be changing them a bit.
  nnl = nl; ncom = com; opi = pi;
  % Now, we find the number of times each node has been used by a path.
  % Call this number 0<=n_i<=k. It is found by simply iterating over each
  % node and setting n_j to the number of times that node occurs in nl.
  nodeusage = zeros(size(pi));
  for i=1:length(nodeusage)
    nodeusage(i) = length(find(nl == i));
  end
  % If solution has no overused nodes, i.e. n_i<2 for all i, the solution
  % primal feasible since all paths are vertex disjoint. In that case, we
  % return the input solution as our primal feasible solution.
  if (max(nodeusage) < 2)
    return
  end
  % If we have overused/infeasible nodes, we save all of them in a vector
```

```
    % in order to iterate over them. We want to return a solution for which
    % length(infeasiblenodes) = 0.
    infeasiblenodes = find(nodeusage >= 2);
    % While infeasiblenodes isn't empty (i.e. its length isn't 0), we eliminate
    % paths passing through nodes found in the list.
    while(~isempty(infeasiblenodes))
        % For every infeasible node, we try to either replace a path with a
        % new one, or simply discard it.
        for i=1:length(infeasiblenodes)
            % First, we reset the nl, com and pi variables by copying the
            % com and nl variables from the current "state" (i.e. the current
            % output variables), and pi from the input variable.
            nl = nnl; com = ncom; pi = opi;
            % We find the first path that passes through the first
            % infeasible node in our list. This code is based on the code
            % in okcom.m, and functions like it. The result is a pair of
            % variables first,last containing the positions of the path in
            % nl, and a pair of variables sl,tl containing the start and
            % end node of the path, respectively.
            node = infeasiblenodes(i);
            first = 0; last = 0; sl = 0; tl = 0;
            for i = 1:size(com,1)
                first = last+1;
                slask = find(nl(last+1:length(nl)) == com(i,1));
                last = slask(1)+first-1;
                if sum(nl(first:last) == node) > 0
                    tl = nl(first);
                    sl = nl(last);
                    break
                end
            end
            % Since we found out path (we always will), we remove it
            % from nl as well as removing its corresponding row in com.
            % If we can replace it, we will.
            nl(first:last) = [];
            com(find(ismember(com, [sl tl], 'rows') == 1), :) = [];
            % Having removed our path from nl, we modify pi by setting an
            % infinite cost for all nodes in nl. This means that when finding
            % an alternative to the path we've removed, the cost of creating
            % a path that isn't vertex-disjoint to the others will be infinite
            % and we can discard such solutions easily.
            pi(nl) = Inf;
            % Using this new pi, we find a new cheapest path between the nodes
            % we removed earlier.
            newnl = gsp(dimX, dimY, pi(:), 1, [tl sl]);
            % If we didn't find a new path, we try again with the next node
            % in the list of infeasible nodes (the modified variables will be
            % reset at the beginning of the next iteration). If we did find
            % a new path, we break out of the loop.
            if length(newnl) < 2 || sum(pi(newnl)) == Inf % If a path wasn't found
                continue                                  % Continue to next iteration
            else                                          % Else
                break                                     % Break out of the loop
            end
```

```
        end
        % Since we're outside the infeasible-node loop, we must have either found a
        % new path, in which case we append it to com/nl, or we didn't, in which case
        % it has been removed from com/nl and we simply update the output variables
        % to match the new set of paths.
        if length(newnl) >= 2 && sum(pi(newnl)) ~= Inf % If a new path was found
            % Append the found path to com/nl
            ncom = [com; sl tl];
            nnl = [nl; flipud(newnl)];
        else
            % Copy the paths without appending anything
            ncom = com;
            nnl = nl;
        end
        % Before the end of the loop, we recalculate the node usage in the same way
        % as before the loop, and update the infeasiblenodes vector to contain the
        % new set of infeasible nodes (which will always be smaller).
        for i=1:length(nodeusage)
            nodeusage(i) = length(find(nl == i));
        end
        infeasiblenodes = find(nodeusage >= 2);
    end
end
```