IE400 Project

IE400 - Principles of Engineering Management
Fall 2019

Project Report

Bartu Atabek - 21602229
Utku Görkem Ertürk - 21502497
Hygerta Imeri - 21603212

December 31, 2019

Table of Contents

1. Introduction

Optimization problems are widely encountered in many areas, but in this report, we will discuss how optimization is required in the electronics manufacturing process. The manufacturing of electronic circuit boards requires a drilling machine that carries out the drilling process, i.e., drilling holes in the pre-defined points in the rectangular-shaped board. Each pair of points is connected through a path made of a conductive material, provided that the head of the drill can move only horizontally or vertically according to the Manhattan distance definition. The drilling process is a costly procedure, and due to a large number of holes that need to be drilled for one board only, minimizing the covered distance needed to drill all of the pre-defined points (including the return to the starting position as well) becomes a crucial objective. In addition to the standard version of the problem, as described so far, our particular problem case includes blocks which constrain the drilling paths. This additional constraint may make some of the paths between pairs of holes unavailable.

The goal of this report as a documentation of our work on the project is to come up with a mathematical model that correctly describes the given problem, provided the constraints, decision variables and parameters of the problem are well-formulated. Also, we aim to recommend the best paths by visualizing the optimal answer obtained by the solver we have utilized.

In the following sections of this report the methodologies that we have used to approach the problem will be explained, followed by the mathematical model. Then, the Solution will be provided, followed by a Discussion section and the Conclusion.

2. Methodology

The first step in our methodology is data pre-processing. We were given the locations as Cartesian coordinates of 50 points (holes in the board) and the locations of the initial points (bottom left corners) together with the dimensions (width, height) of the rectangular blocks.

Firstly, the data is read from the given files, then all the possible paths (taking into consideration the blocked ways) are calculated according to Manhattan distance. The paths are found and their lengths are stored.

Having processed the data, we used the Traveling Salesman Problem (TSP) approach to find the shortest path (i.e., the path with the optimal cost).

Thinking of the points on the given 2D matrix-like board configuration as nodes of an undirected graph, where the distance between two points is the same in either direction, we can say that the given problem boils down to the Symmetric Traveling Salesman Problem. Since in the Traveling Salesman Problem we look for the shortest closed path, in our problem this translates to visiting (drilling) all points and returning to the starting point as well. Due to this resemblance, we decided to use the TSP model approach to solve the given problem. The solution to this problem is written in MATLAB programming language and the solver used is the mixed-integer linear programming solver 'intlinprog' [1].

3. Mathematical Model

In this section the mathematical model will be described.

3.1 Parameters

The parameters of this problem are paths between Cartesian coordinates and their distances (i.e. there is no block in the shortest Manhattan distance between two points). The information about paths and distances is retrieved from the already pre-processed data.

3.2 Decision Variables

The decision variable in our problem is a binary decision variable to which we assign the value "1" whenever there is a path between two points in the tour and we assign "0" otherwise.

$$X_{i,J} = \begin{cases} 1 & if\ there\ is\ a\ path\ from\ point\ i\ to\ point\ J \\ 0 & otherwise \end{cases}$$

3.3 Objective Function and Constraints

The objective function of this problem is to minimize the total cost of drilling all of the points on the circuit board during the manufacturing process. The total cost of drilling means the total distance covered to visit all points taking into consideration all of the following constraints.

The first constraint guarantees that there is just one path going to (pointing towards) each point. The second constraint guarantees that there is just one path leaving each point. Altogether these constraints (1 and 2) ensure that there are 2 paths for each of the points. The third constraint means that there is no path from point i to itself. The last constraint guarantees the integrality of the solution.

The objective function of this problem and constraints are as follows:

$$\min \sum_{J}^{n} \sum_{i}^{n} c_{iJ} X_{iJ}$$

subject to

$$\sum_{J=1}^{n} X_{iJ} = 1 \qquad \forall i$$

$$\sum_{i=1}^{n} X_{iJ} = 1 \qquad \forall J$$

$$X_{ii} = 0 \qquad \forall i$$
$$X_{iJ} = \{0, 1\} \qquad \forall i, J$$

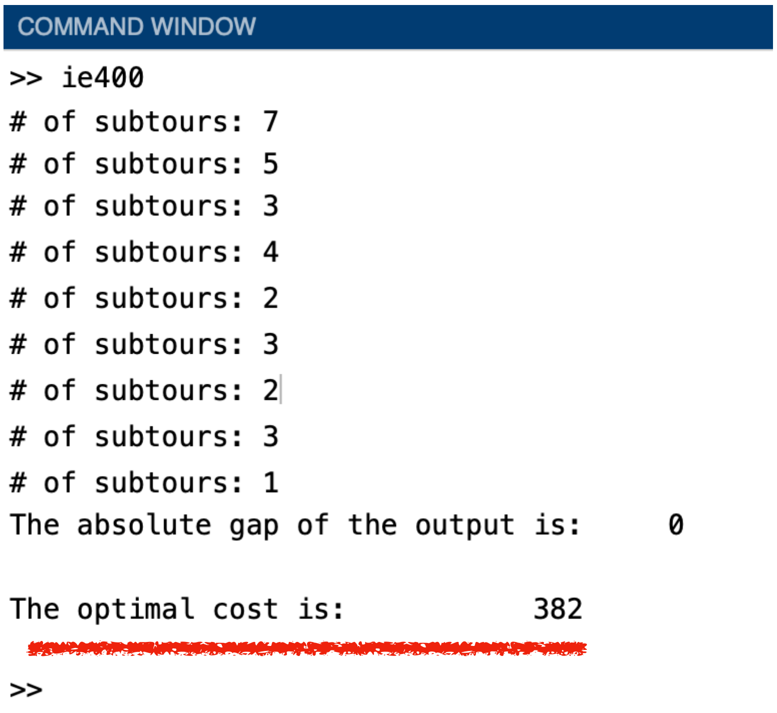
In order to eliminate ‘sub-tours/subpaths’ (until only one sub-tour/subpath remains), another constraint was added

$$\sum_{i \in S}^{n} \sum_{J \notin S}^{n} X_{iJ} \geq 1 \qquad \forall S \subseteq N,\ S \neq \phi$$

This guarantees that there will be no cyclic path (a subpath) in the solution which is found in fewer iterations.

4. Solution (Findings/Results)

The solutions to the given problem are visually presented in Fig. 3 and Fig. 4 below and we have checked the optimality of our solution by examining the structure of the output result as follows: disp(output. absolutegap) as it is shown in Fig.1 and Fig. 2. The smaller the value that comes from *disp(output. absolutegap)* the better the solution. We got an absolute gap equal to zero, confirming thus that our obtained solution is indeed optimal. The value of the optimal solution for the provided dataset of this project is “**382"**. We found the distances/paths between points in two ways, In both of them there is no change in the optimal value of the objective function, neither in the optimality of the solution, yet there are some changes in number of sub-tours and number of iterations of the problem until number of sub-tours becomes 1. In the second case, paths are found using the Breadth-First-Search, more iterations are encountered and slightly longer time is needed for the results to be calculated.

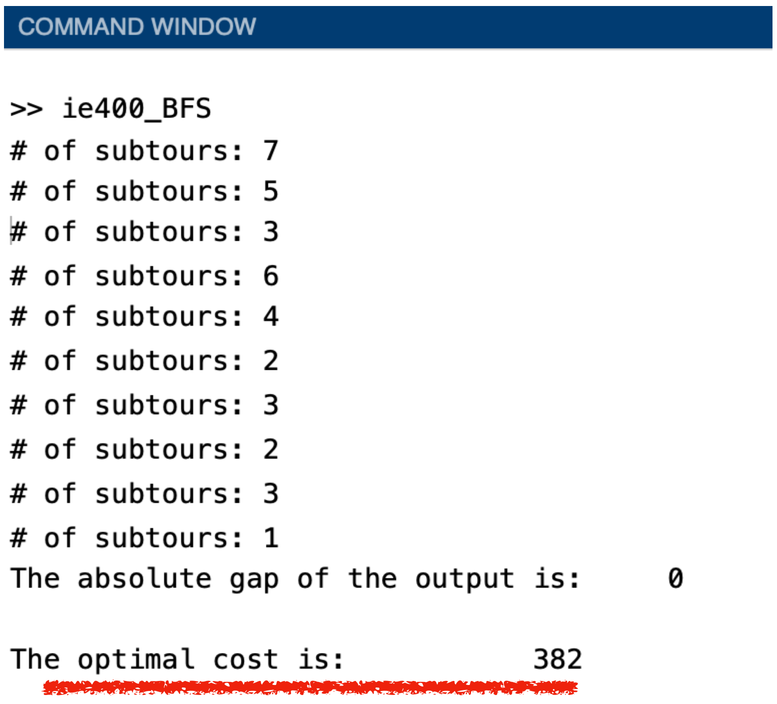


```
>> ie400
# of subtours: 7
# of subtours: 5
# of subtours: 3
# of subtours: 4
# of subtours: 2
# of subtours: 3
# of subtours: 2
# of subtours: 3
# of subtours: 1
The absolute gap of the output is:     0

The optimal cost is:          382

>>
```

Fig. 1. The output (1st implementation - DFS)



```
>> ie400_BFS
# of subtours: 7
# of subtours: 5
# of subtours: 3
# of subtours: 6
# of subtours: 4
# of subtours: 2
# of subtours: 3
# of subtours: 2
# of subtours: 3
# of subtours: 1
The absolute gap of the output is:      0

The optimal cost is:            382
```

Fig. 2. The output (2nd implementation - BFS)

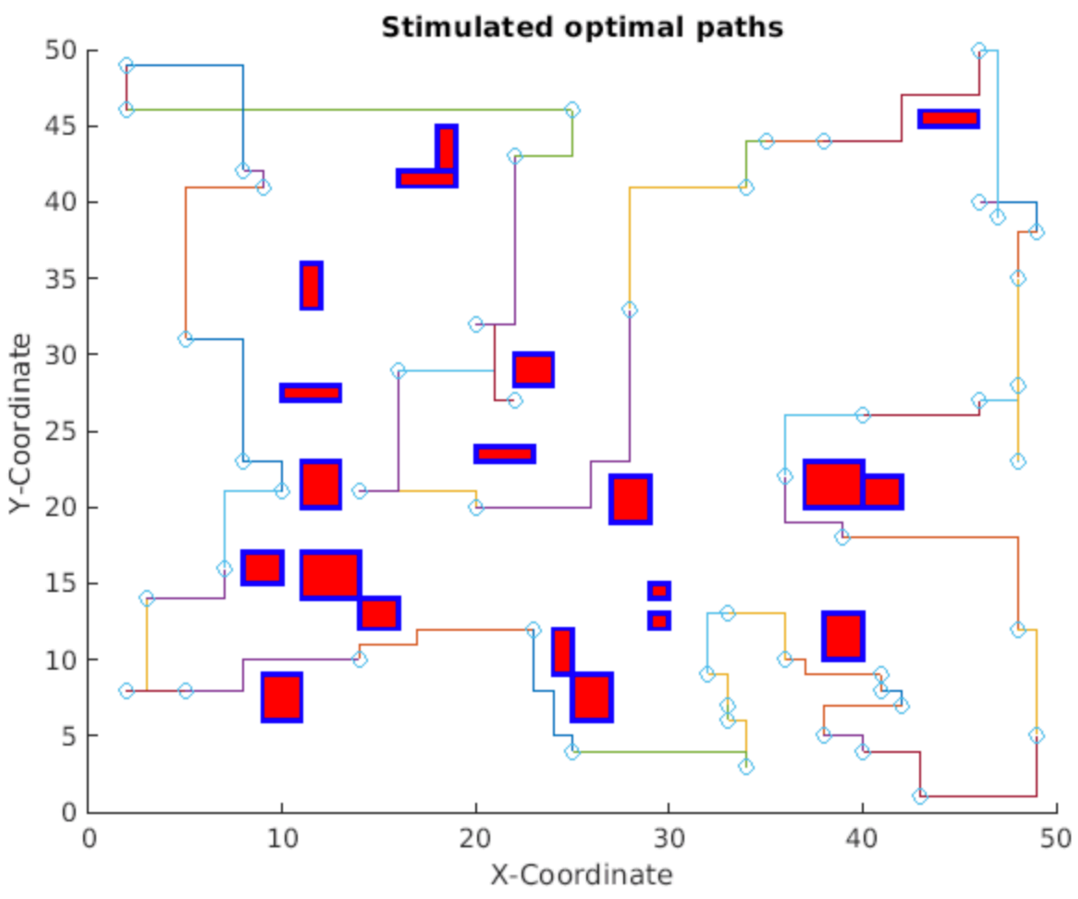


Fig. 3. The best path plan stimulated for the 1st implementation

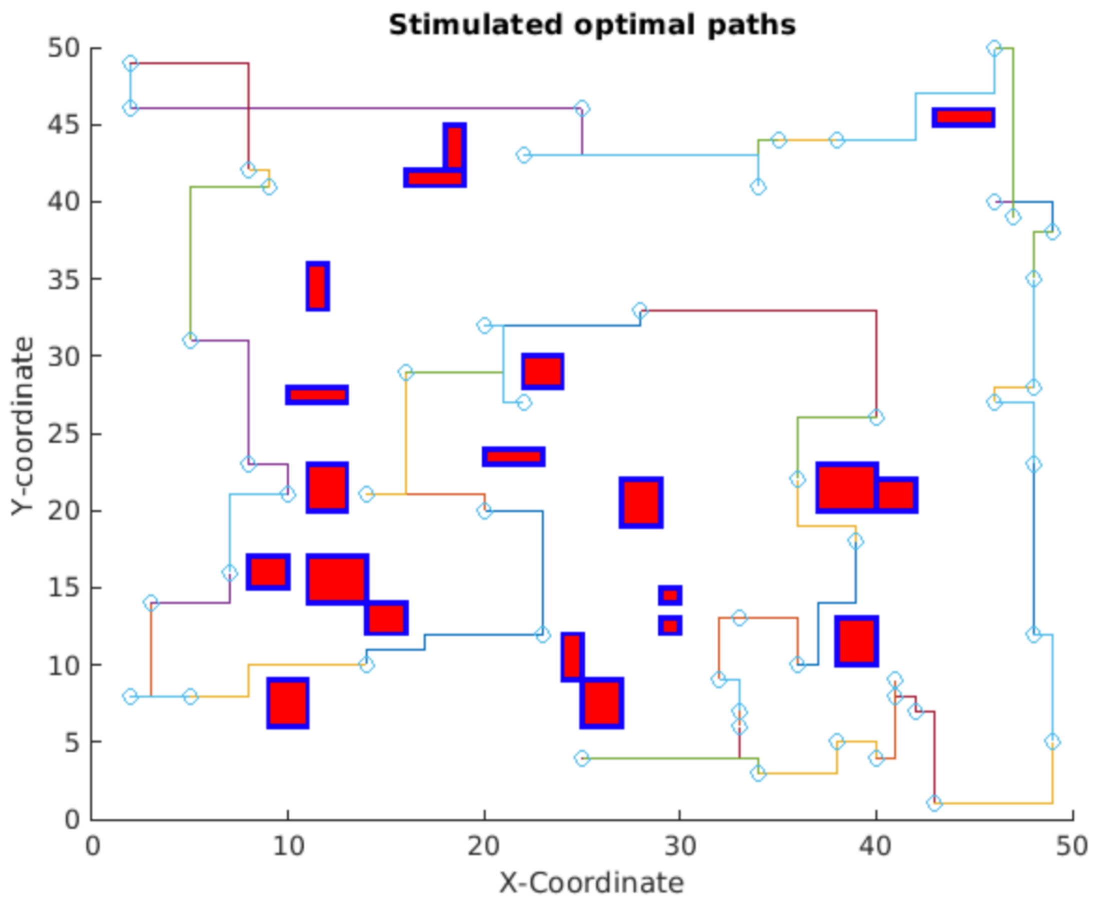


Fig. 4. The best path plan stimulated by the solver (BFS- 2nd implementation)

5. Discussion & Conclusions

There might be several ways of solving one problem, and even though all of them yield the same optimal value of the objective function, they differ in terms of algorithmic runtime. After the data with the added block constraints was preprocessed in a way such that all possible paths connecting the points were obtained, we could see how the problem we were left with became too similar to the very well-known Traveling Salesman Problem. Then TSP approach and 'intlineprog' solver provided in MATLAB handled the rest of the problem.

References

[1] Mathworks.com. (2019). *Traveling Salesman Problem: Solver-Based-MATLAB & Simulink*. [online] Available at: https://www.mathworks.com/help/optim/ug/travelling-salesman-problem.html [Accessed 28 Dec. 2019].

Appendix

Matlab Code (DFS based conditioned implementation)

```
%Read data
blocks = readmatrix("blocks.txt");
points = readmatrix("points.txt");

idxs = nchoosek(1:size(points,1),2);
paths = cell(size(points,1),1);
pathLengths = zeros(length(idxs),1);
counter = 1;
for i = 1:size(points,1)
    for j = i+1:size(points,1)[~,paths{counter},pathLengths(counter)] =
findPath(points(i,:),points(j,:),blocks);
        counter = counter + 1;
    end
end

idxs(~pathLengths,:)= [];
paths(~pathLengths) = [];
pathLengths(~pathLengths) = [];
dist = pathLengths;
lendist = length(dist);

Aeq = spones(1:length(idxs)); % Adds up the number of trips
beq = size(points,1);


Aeq=[Aeq;spalloc(size(points,1),length(idxs),size(points,1)*(size(points,1)-1))]; %
allocate a sparse matrix
for ii = 1:size(points,1)
    whichIdxs = (idxs == ii); % find the trips that include stop ii
    whichIdxs = sparse(sum(whichIdxs,2)); % include trips where ii is at either end
    Aeq(ii+1,:) = whichIdxs'; % include in the constraint matrix
end
beq = [beq; 2*ones(size(points,1),1)];

intcon = 1:lendist;
lb = zeros(lendist,1);
ub = ones(lendist,1);

opts = optimoptions('intlinprog','Display','off');
[x_tsp,costopt,exitflag,output]=intlinprog(dist,intcon,[],[],Aeq,beq,lb,ub,opts);

tours = detectSubtours(x_tsp,idxs);
numtours = length(tours); % number of subtours
fprintf('# of subtours: %d\n',numtours);
```

```
A = spalloc(0,lendist,0); % Allocate a sparse linear inequality constraint matrix
b = [];
while numtours > 1 % repeat until there is just one subtour
    % Add the subtour constraints
    b = [b;zeros(numtours,1)]; % allocate b
    A = [A;spalloc(numtours,lendist,size(points,1))]; % a guess at how many
nonzeros to allocate
    for ii = 1:numtours
        rowIdx = size(A,1)+1; % Counter for indexing
        subTourIdx = tours{ii}; % Extract the current subtour
%          The next lines find all of the variables associated with the
%          particular subtour, then add an inequality constraint to prohibit
%          that subtour and all subtours that use those stops.
        variations = nchoosek(1:length(subTourIdx),2);
        for jj = 1:length(variations)
            whichVar = (sum(idxs==subTourIdx(variations(jj,1)),2)) & ...
                       (sum(idxs==subTourIdx(variations(jj,2)),2));
            A(rowIdx,whichVar) = 1;
        end
        b(rowIdx) = length(subTourIdx)-1; % One less trip than subtourstops
    end

    % Try to optimize again
[x_tsp,costopt,exitflag,output]=intlinprog(dist,intcon,A,b,Aeq,beq,lb,ub,opts);

    % How many subtours this time?
    tours = detectSubtours(x_tsp,idxs);
    numtours = length(tours); % number of subtours
    fprintf('# of subtours: %d\n',numtours);
end

optimizedPaths = paths(find(x_tsp));
fprintf('The absolute gap of the output is:');
disp(output.absolutegap);
fprintf('The optimal cost is: ');
format shortG
disp(costopt);
figure;
hold on

for i = 1:size(points,1)
    xs = [];
    ys = [];
    currpath = optimizedPaths{i};

    for j = 1:size(currpath,1)
        xs = [xs currpath(j,1)];
        ys = [ys currpath(j,2)];
        legend
        plot(xs,ys);
    end
end
plot(points(:,1), points(:,2), 'o');
for i = 1:size(blocks,1)
    rectangle('Position', blocks(i,:), 'EdgeColor', 'b', 'FaceColor', 'r',
'LineWidth', 2);
end



function [hasPath,path,pathLength] = findPath(S,F,blocks)
    path = [];
    visitedMat = zeros(abs(S-F)+1);
    pathLength = 0;
    hasPath = findPathHelper(S,0);
    function x = findPathHelper(N,found)
        x = found;
        if(~isBlocked(blocks,N) && isInBoundary(S,F,N) && ~found)
            [visited,visitedMat] = isVisitedMark(S,F,visitedMat,N);
            if(visited)
                return;
            end
            if(N == F)
                x = 1;
                path = [path ; N];
                return;
            end
            if(S(2)<=F(2))
                x = findPathHelper(N + [0 1],x);
            end
            if(S(1)<=F(1))
                x = findPathHelper(N + [1 0],x);
            end
            if(S(2)>=F(2))
                x = findPathHelper(N + [0 -1],x);
            end
            if(S(1)>=S(1))
                x = findPathHelper(N + [-1 0],x);
            end

            if(x)
                pathLength = pathLength + 1;
                path = [path ; N];
            end
        end
    end
end

function x = isBlocked(blocks,N)
    for i = 1:size(blocks,1)
        xmin = blocks(i,1);
        xmax = blocks(i,1) + blocks(i,3);
        ymin = blocks(i,2);
        ymax = blocks(i,2) + blocks(i,4);

        if(xmin <= N(1) && N(1) <= xmax && ymin <= N(2) && N(2) <= ymax)
            x = 1;
            return
        end
    end
    x = 0;
end
function x = isInBoundary(S,F,N)
    xmin = min(S(1),F(1));
    xmax = max(S(1),F(1));

    ymin = min(S(2),F(2));
    ymax = max(S(2),F(2));
    if(xmin <= N(1) && N(1) <= xmax && ymin <= N(2) && N(2) <= ymax)
        x = 1;
    else
        x = 0;
    end
end

function [x,visitedMat] = isVisitedMark(S,F,visitedMat,N)
    xmin = min(S(1),F(1));
    ymin = min(S(2),F(2));
    index = N -[xmin ymin];
    x = visitedMat(index(1) + 1,index(2) + 1);
    visitedMat(index(1) + 1,index(2) + 1) = 1;
end


function subTours = detectSubtours(x,idxs)
% Returns a cell array of subtours. The first subtour is the first row of x, etc.

%   Copyright 2014 The MathWorks, Inc.

x = round(x); % correct for not-exactly integers
r = find(x); % indices of the trips that exist in the solution
substuff = idxs(r,:); % the collection of node pairs in the solution
unvisited = ones(length(r),1); % keep track of places not yet visited
curr = 1; % subtour we are evaluating
startour = find(unvisited,1); % first unvisited trip
    while ~isempty(startour)
        home = substuff(startour,1); % starting point of subtour
        nextpt = substuff(startour,2); % next point of tour
        visited = nextpt; unvisited(startour) = 0; % update unvisited points
        while nextpt ~= home
            % Find the other trips that starts at nextpt
            [srow,scol] = find(substuff == nextpt);
            % Find just the new trip
            trow = srow(srow ~= startour);
            scol = 3-scol(trow == srow); % turn 1 into 2 and 2 into 1
            startour = trow; % the new place on the subtour
            nextpt = substuff(startour,scol); % the point not where we came from
            visited = [visited,nextpt]; % update nodes on the subtour
            unvisited(startour) = 0; % update unvisited
        end
        subTours{curr} = visited; % store in cell array
        curr = curr + 1; % next subtour
        startour = find(unvisited,1); % first unvisited trip
    end
end
```

Matlab Code(BFS based implementation)

```
%Read data
blocks = readmatrix("blocks.txt");
points = readmatrix("points.txt");

idxs = nchoosek(1:size(points,1),2);
paths = cell(size(points,1),1);
pathLengths = zeros(length(idxs),1);
counter = 1;
for i = 1:size(points,1)
    for j = i+1:size(points,1)
        [~,paths{counter},pathLengths(counter)] = 
findPath(points(i,:),points(j,:),blocks);
        counter = counter + 1;
    end
end

idxs(~pathLengths,:)= [];
paths(~pathLengths) = [];
pathLengths(~pathLengths) = [];
dist = pathLengths;
lendist = length(dist);
```

```
Aeq = spones(1:length(idxs)); % Adds up the number of trips
beq = size(points,1);


Aeq = [Aeq;spalloc(size(points,1),length(idxs),size(points,1)*(size(points,1)-1))];
% allocate a sparse matrix
for ii = 1:size(points,1)
    whichIdxs = (idxs == ii); % find the trips that include stop ii
    whichIdxs = sparse(sum(whichIdxs,2)); % include trips where ii is at either end
    Aeq(ii+1,:) = whichIdxs'; % include in the constraint matrix
end
beq = [beq; 2*ones(size(points,1),1)];

intcon = 1:lendist;
lb = zeros(lendist,1);
ub = ones(lendist,1);

opts = optimoptions('intlinprog','Display','off');
[x_tsp,costopt,exitflag,output] = intlinprog(dist,intcon,[],[],Aeq,beq,lb,ub,opts);

tours = detectSubtours(x_tsp,idxs);
numtours = length(tours); % number of subtours
fprintf('# of subtours: %d\n',numtours);

A = spalloc(0,lendist,0); % Allocate a sparse linear inequality constraint matrix
b = [];
while numtours > 1 % repeat until there is just one subtour
    % Add the subtour constraints
    b = [b;zeros(numtours,1)]; % allocate b
    A = [A;spalloc(numtours,lendist,size(points,1))]; % a guess at how many
nonzeros to allocate
    for ii = 1:numtours
        rowIdx = size(A,1)+1; % Counter for indexing
        subTourIdx = tours{ii}; % Extract the current subtour
%         The next lines find all of the variables associated with the
%         particular subtour, then add an inequality constraint to prohibit
%         that subtour and all subtours that use those stops.
        variations = nchoosek(1:length(subTourIdx),2);
        for jj = 1:length(variations)
            whichVar = (sum(idxs==subTourIdx(variations(jj,1)),2)) & ...
                       (sum(idxs==subTourIdx(variations(jj,2)),2));
            A(rowIdx,whichVar) = 1;
        end
        b(rowIdx) = length(subTourIdx)-1; % One less trip than subtour stops
    end

    % Try to optimize again
    [x_tsp,costopt,exitflag,output] =
intlinprog(dist,intcon,A,b,Aeq,beq,lb,ub,opts);


    % How many subtours this time?
    tours = detectSubtours(x_tsp,idxs);
    numtours = length(tours); % number of subtours
    fprintf('# of subtours: %d\n',numtours);
end

optimizedPaths = paths(find(x_tsp));
fprintf('The absolute gap of the output is:');
disp(output.absolutegap);
fprintf('The optimal cost is: ');
format shortG
disp(costopt);
```

```
figure;
hold on

for i = 1:size(points,1)
    xs = [];
    ys = [];
    currpath = optimizedPaths{i};

    for j = 1:size(currpath,1)
        xs = [xs currpath(j,1)];
        ys = [ys currpath(j,2)];
        legend
        plot(xs,ys);
    end
end
plot(points(:,1), points(:,2), 'o');
for i = 1:size(blocks,1)
    rectangle('Position', blocks(i,:), 'EdgeColor', 'b', 'FaceColor', 'r',
'LineWidth', 4);
end


function [hasPath,path,pathLength] = findPath(S,F,blocks)
    import java.util.LinkedList
    q = LinkedList();
    visitedMat = zeros(abs(S-F)+1);
    previousMat = cell(abs(S-F)+1);
    q.add(S);
    index = calculateOriginIndex(S,F,S);
    visitedMat(index(1),index(2)) = 1;
    while(~q.isEmpty())
        N = q.remove()';
        if(N == F)
            break;
        end
        possMoves = [N + [0 1] ; N + [1 0] ; N + [0 -1] ; N + [-1 0]];
        for i = 1:size(possMoves,1)
            index = calculateOriginIndex(S,F,possMoves(i,:));

            if(~isBlocked(blocks,possMoves(i,:)) &&
isInBoundary(S,F,possMoves(i,:)) && ~visitedMat(index(1),index(2)))

                q.add(possMoves(i,:));
                visitedMat(index(1),index(2)) = 1;
                previousMat{index(1),index(2)} = N;
            end
        end
    end
   [hasPath,pathLength,path] = findPathHelper(S,F,previousMat);
end

function x = isBlocked(blocks,N)
    for i = 1:size(blocks,1)
        xmin = blocks(i,1);
        xmax = blocks(i,1) + blocks(i,3);
        ymin = blocks(i,2);
        ymax = blocks(i,2) + blocks(i,4);

        if(xmin <= N(1) && N(1) <= xmax && ymin <= N(2) && N(2) <= ymax)
            x = 1;
            return
        end
    end
    x = 0;
```

```
end
function x = isInBoundary(S,F,N)
    xmin = min(S(1),F(1));
    xmax = max(S(1),F(1));

    ymin = min(S(2),F(2));
    ymax = max(S(2),F(2));
    if(xmin <= N(1) && N(1) <= xmax && ymin <= N(2) && N(2) <= ymax)
        x = 1;
    else
        x = 0;
    end
end

function index = calculateOriginIndex(S,F,N)
    xmin = min(S(1),F(1));
    ymin = min(S(2),F(2));
    index = N -[xmin ymin]+[1 1];
end

function [hasPath,pathLength,path] = findPathHelper(S,F,previousMat)
    path = [];
    pathLength = 0;
    hasPath = 1;
    if (S == F)
        path = [path ; S];
        return
    end
    index = calculateOriginIndex(S,F,F);
    if(isempty(previousMat{index(1),index(2)}))
        hasPath = 0;
        return
    end
    path = [path ; F];
    desiredDistance = sum(abs(S-F));
    while(~isempty(previousMat{index(1),index(2)}))
        N = previousMat{index(1),index(2)};
        path = [path ; N];
        pathLength = pathLength + 1;
        index = calculateOriginIndex(S,F,N);
        if(pathLength > desiredDistance)
            path = [];
            pathLength = 0;
            hasPath = 0;
            break;
        end
    end
end

function subTours = detectSubtours(x,idxs)
    % Returns a cell array of subtours. The first subtour is the first row of x,
etc.
    % Copyright 2014 The MathWorks, Inc.
    x = round(x); % correct for not-exactly integers
    r = find(x); % indices of the trips that exist in the solution
    substuff = idxs(r,:); % the collection of node pairs in the solution
    unvisited = ones(length(r),1); % keep track of places not yet visited
    curr = 1; % subtour we are evaluating
    startour = find(unvisited,1); % first unvisited trip
    while ~isempty(startour)
        home = substuff(startour,1); % starting point of subtour
        nextpt = substuff(startour,2); % next point of tour
        visited = nextpt; unvisited(startour) = 0; % update unvisited points
        while nextpt ~= home
```

```
            % Find the other trips that starts at nextpt
            [srow,scol] = find(substuff == nextpt);
            % Find just the new trip
            trow = srow(srow ~= startour);
            scol = 3-scol(trow == srow); % turn 1 into 2 and 2 into 1
            startour = trow; % the new place on the subtour
            nextpt = substuff(startour,scol); % the point not where we came from
            visited = [visited,nextpt]; % update nodes on the subtour
            unvisited(startour) = 0; % update unvisited
        end
        subTours{curr} = visited; % store in cell array
        curr = curr + 1; % next subtour
        startour = find(unvisited,1); % first unvisited trip
    end
end
```