

Caleb Logemann

MATH 566 Discrete Optimization

Midterm I

1. Suppose you are making a schedule for an airport. There are n arriving flights. Every airplane j has a possible time arrival in interval $[a_j, b_j]$ (plane can fly faster or slower). Determine the actual arrival schedule for each airplane such that the smallest gap between consecutive flights is maximized and for all j , airplane j arrives before $j + 1$. Formulate a linear program that solves the problem.

(Example: Suppose there are three airplanes. They have arrival intervals $[1, 5]$, $[2, 7]$, $[6, 7]$. Then we can assign arrival times to the airplanes, for example 2, 4.5, 6.2. The smallest gap in this schedule is 1.7 between the second and third airplane. The number 1.7 is the number we want to maximize. Notice that we do not allow schedule 4, 2, 7, where the first airplane arrives AFTER the second one although it would be feasible with respect to $[a_i, b_i]$ s (it is easier to solve if the order is fixed).)

First I will define a new set of variables, t_j for $1 \leq j \leq n$, to be the time that plane j arrives at the airport. Clearly we must have the constraints

$$\begin{aligned} t_j &\geq a_j \\ t_j &\leq b_j \end{aligned}$$

for all j , such that $1 \leq j \leq n$. These constraints force each plane to arrive inside of the allowed interval $[a_j, b_j]$. Also if the order that the planes arrive is to be enforced, the constraints

$$t_{j+1} - t_j \geq 0$$

are required for $1 \leq j \leq n - 1$. Lastly we need to set the objective function for this linear program. Our objective function needs to maximize the smallest time gap $t_{j+1} - t_j$ or in other words $\max \min_{1 \leq j \leq n-1} (t_{j+1} - t_j)$. I am going to create another variable m to be this minimum value, that is

$$m = \min_{1 \leq j \leq n-1} (t_{j+1} - t_j)$$

In this case the objective function just becomes $\max m$. However we need constraints to insure that m does in fact equal the minimum value. I will use the following constraints

$$m \leq t_{j+1} - t_j$$

for all j such that $1 \leq j \leq n - 1$. These constraints guarantee that

$$m \leq \min_{1 \leq j \leq n-1} (t_{j+1} - t_j).$$

With the addition of the objective function maximizing m , we will achieve equality. Therefore the entire linear program that solves this problem is

$$(P) = \begin{cases} \text{maximize} & m \\ \text{subject to} & t_j \geq a_j \quad 1 \leq j \leq n \\ & t_j \leq b_j \quad 1 \leq j \leq n \\ & t_{j+1} - t_j \geq 0 \quad 1 \leq j \leq n - 1 \\ & m \leq t_{j+1} - t_j \quad 1 \leq j \leq n - 1 \end{cases}$$

2. Solve the following linear program (P) using simplex method.

$$(P) = \begin{cases} \text{maximize} & x_1 + x_2 \\ \text{subject to} & x_1 \leq 1 \\ & -x_1 + x_2 \leq 1 \\ & x_1, x_2 \geq 0 \end{cases}$$

Check your solution using computer program (APMonitor, Sage,...). Plot the set of feasible solutions and mark the optimum. Solving using simplex method means make the sequence of simplex tables.

3. Consider the following algorithm. Input is a connected graph $G = (V, E)$ and a cost function $c : E \rightarrow \mathbb{R}$. Start with H being a copy of G . First, the edges E are ordered such that $c(e_1) \geq c(e_2) \geq \dots \geq c(e_m)$. Then process edges one by one according to the ordering. Processing edge e_i means looking if $H - e_i$ connected. If $H - e_i$ is connected, then e_i is removed from H . Otherwise e_i is kept in H . After all edges are processed, the resulting H is the output. Now you can pick what to do. Either a) or b):

(a) Implement the algorithm and use as inputs the same graph we used for the minimum spanning tree

(b) Prove that the algorithm produces minimum spanning tree.

I chose to do part (a). The following script creates random graphs and implements the given algorithm in the function minimumSpanningTree.

```
import itertools as it
load('breadthFirstSearch.sage')

# This plots vertices as red dots and blue edges connecting them
def plot_vertices_edges(vertices, edges):
    drawing = line([])
    for x in vertices:
        drawing = drawing + disk(x, 0.1, (0, 2*pi), color='red')
    for e in edges:
        drawing = drawing + line([vertices[e[0]], vertices[e[1]]])
    drawing.show()

# Generate 10 random vertices in 10x10 grid
def generate_random_vertices():
    vertexList = []
    for i in range(10):
        vertexList.append((random()*10, random()*10))
    return vertexList

def generateEdgeList(numVertices):
    return list(it.combinations(range(numVertices), 2))

def generateCostList(vertexList, edgeList):
    costList = []
    for edge in edgeList:
        u = vertexList[edge[0]]
        v = vertexList[edge[1]]
        costList.append(sqrt((u[0]-v[0])**2 + (u[1]-v[1])**2))
    return costList

def minimumSpanningTree(vertexList, edgeList, costList):
    numVertices = len(vertexList)
    numEdges = len(edgeList)
```

```

# sort edges by cost
s = sorted(zip(edgeList, costList), key=lambda pair:pair[1], reverse
    ↪ =True)
edgeList = [x for (x, y) in s]
costList = [y for (x, y) in s]

treeEdgeList = list(edgeList)

for edge in edgeList:
    treeEdgeList.remove(edge)
    # if graph without edge is not connected
    if max(breadthFirstSearch(vertexList, treeEdgeList, vertexList
        ↪ [0]).values()) == oo:
        # if not connected add edge back into treeEdgeList
        treeEdgeList.append(edge)

return treeEdgeList

# generate random graph
vertexList = generate_random_vertices()
edgeList = generateEdgeList(len(vertexList))
costList = generateCostList(vertexList, edgeList)

# find minimum spanning tree
edges = minimumSpanningTree(range(10), edgeList, costList)

# plot minimum spanning tree
plot_vertices_edges(vertexList, edges)

```

This function requires a breadthSearchFirst algorithm which I implemented as follows.

```

import Queue
def breadthFirstSearch(vertexList, edgeList, rootVertex):
    parentDict = dict()
    distanceDict = dict()
    for vertex in vertexList:
        parentDict[vertex] = None
        distanceDict[vertex] = oo

    # create queue to hold nodes
    q = Queue.Queue()

    distanceDict[rootVertex] = 0
    q.put(rootVertex)

    while not q.empty():
        currentVertex = q.get()
        for edge in edgeList:
            if edge[0] == currentVertex:
                adjacentVertex = edge[1]

```

```

elif edge[1] == currentVertex:
    adjacentVertex = edge[0]
else:
    continue

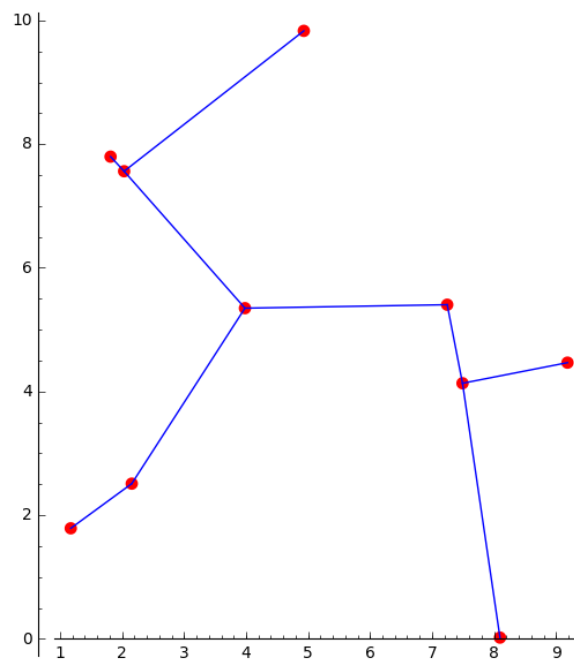
# if we haven't reached adjacentVertex yet
if distanceDict[adjacentVertex] == oo:
    distanceDict[adjacentVertex] = distanceDict[
        ↪ currentVertex] + 1
    parentDict[adjacentVertex] = currentVertex
    q.put(adjacentVertex)

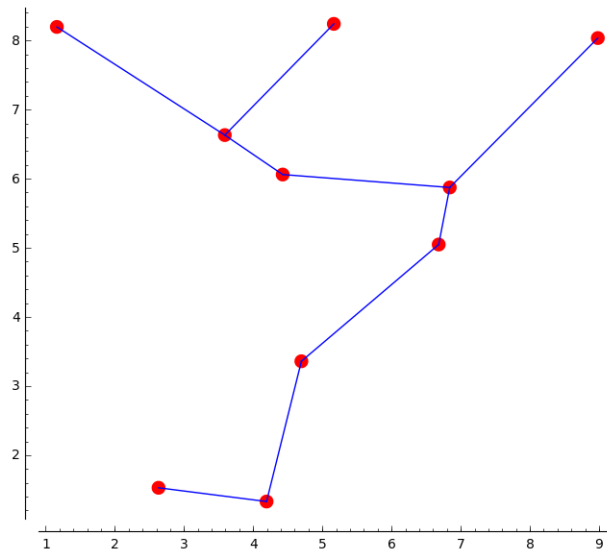
return distanceDict

```

This function checks to see if a graph is connected. If a graph isn't connected it will return with an infinite distance from the root vertex to some other vertex.

The following two plots are the results on running the initial script twice. They show that the algorithm does in fact find the minimum spanning tree.





4. Consider the following problem. Input is a connected graph $G = (V, E)$ and a cost function $c : E \rightarrow \mathbb{R}$. Let T be a spanning tree of G . The cost of T is defined as the largest cost of an edge in T :

$$c(T) = \max\{c(e) : e \in E(T)\}.$$

Problem is to find a minimum spanning tree with respect to c
 Do both a) and b):

- (a) Formulate the problem using integer programming
- (b) Find an algorithm for solving this problem in polynomial time and prove its correctness.