Boltzmann Machine
Application for Traveling Salesman problem

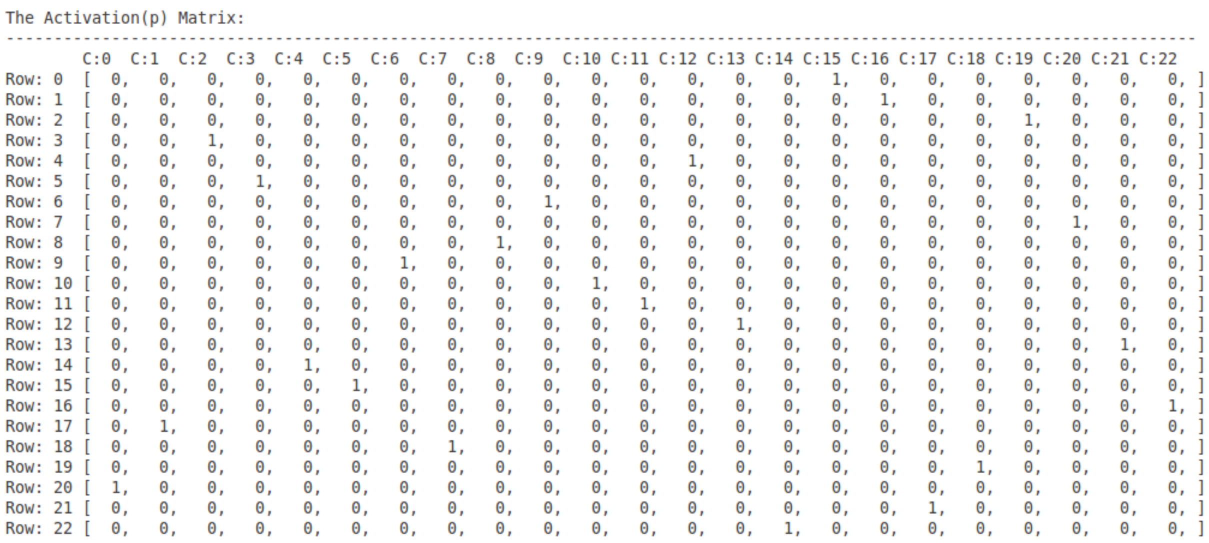
Michal Frystacky

May 8, 2011

Discussion

The traveling salesman problem(TSP) was computed using a modified version of the Boltzmann machine. By having a matrix which represents the distances between cities and having an initially randomly generated binary activation matrix. Having this matrix and a user defined temperature, bonus and penalty we compute the delta c and the probability of acceptance. We then compare the probability of acceptance with a randomly generated number and if the number is less than the probability of acceptance we change that element in the activation matrix to the opposite. We repeat this process for every element in the activation matrix randomly and then decrease the temperature by 5 per cent. We continue doing this until the temperature falls bellow a user defined threshold. Then we acquire our local minimum distance traveled and the activation matrix. This process is repeated 10,000 times and from this we pick our best result which is our outputted minimum.

results



```
The Activation(p) Matrix:
-----------------------------------------------------------------------------------------------------------------------
        C:0  C:1  C:2  C:3  C:4  C:5  C:6  C:7  C:8  C:9  C:10 C:11 C:12 C:13 C:14 C:15 C:16 C:17 C:18 C:19 C:20 C:21 C:22
Row: 0  [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   1,   0,   0,   0,   0,   0,   0,   0, ]
Row: 1  [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   1,   0,   0,   0,   0,   0,   0, ]
Row: 2  [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   1,   0,   0,   0, ]
Row: 3  [  0,   0,   1,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0, ]
Row: 4  [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   1,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0, ]
Row: 5  [  0,   0,   0,   1,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0, ]
Row: 6  [  0,   0,   0,   0,   0,   0,   0,   0,   0,   1,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0, ]
Row: 7  [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   1,   0,   0, ]
Row: 8  [  0,   0,   0,   0,   0,   0,   0,   0,   1,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0, ]
Row: 9  [  0,   0,   0,   0,   0,   0,   1,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0, ]
Row: 10 [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   1,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0, ]
Row: 11 [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   1,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0, ]
Row: 12 [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   1,   0,   0,   0,   0,   0,   0,   0,   0,   0, ]
Row: 13 [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   1,   0, ]
Row: 14 [  0,   0,   0,   0,   1,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0, ]
Row: 15 [  0,   0,   0,   0,   0,   1,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0, ]
Row: 16 [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   1, ]
Row: 17 [  0,   1,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0, ]
Row: 18 [  0,   0,   0,   0,   0,   0,   0,   1,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0, ]
Row: 19 [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   1,   0,   0,   0,   0, ]
Row: 20 [  1,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0, ]
Row: 21 [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   1,   0,   0,   0,   0,   0, ]
Row: 22 [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   1,   0,   0,   0,   0,   0,   0,   0,   0, ]
```

The minimum distance traveled: **3107**

Improvements

Currently it takes about 30 minutes to calculate the best minimum distance traveled of the TSP. The algorithm seems to have little tolerance for certain conditions which leave one or more rows with no 1's, this poses a problem as it gives a false positive for a minimum distance. This issue seems to occur much more frequently when we include a wrap around in the algorithm, so this part was commented out. Having a single activation matrix and testing out many different bonuses, penalties and temperatures to see which combination improves our minimum distance would also improve this algorithm. Another improvement that may significantly improve the time of the algorithm is generating less ones when creating the activation matrix, having too many useless ones creates more calculations for the Boltsmann Machine(BM) to compute which can become unnecessary, it also seems to improve the minimum distance found when generating slightly less ones about .35 - .45 rather than having a .5 chance of generating a 1. This algorithm would also be a good candidate for penalization, due to much of the work in the algorithm being independent (there is very little information sharing due to each processor only having to find a local minimum and sharing that with the master processor), which would increase the time and/or accuracy of the minimum found.

Code

```
'''
Created on May 7, 2011

@author: Michal Frystacky
'''
import os,random, math

class BasicMatrix:

    def __init__(self):
        self.name = ""
        self.data =
            [[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[]]
        for i in range(23):
            for j in range(23):
                self.data[i].append(0)

    def __str__(self):
        format = "%3i, "
        str = "%s\n" % ("The %s:" % self.name )
        str += '-' * 126
        str += "\n"
        str += " " * 8
        for i in range(23):
            str += "C:%-2i " % i
        str += "\n"
        i = 0
        for row in self.data:
            str += "Row: %-2i [" % i
            i += 1
            for j in range (len(row)):
                str += format % row[j]
            str += "]\n"
        return str

class Matrix(BasicMatrix):

    def __init__(self, fileName):
        """
        credit for help with opening files properly
        @credit:http://stackoverflow.com/questions/4060221/how-to-reliably-open-a-file-in-the-same-di
        """
        self.name = "Distance Matrix"
        self.data =
            [[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[]]
        self.__location__ = os.path.realpath(os.path.join(os.getcwd(),
            os.path.dirname(__file__)))
        f = open(os.path.join(self.__location__, fileName))
        for i in range(23):
            for j in range(23):
                temp = f.readline()
                if (temp == ' '): #We do not add the EOL character
                    pass
                else:
                    self.data[i].append(int(temp))

class ActivationMatrix(BasicMatrix):

    def init(self, b, p, t):
        #BONUS PENALTY TEMPERATURE
        self.bonus = b
        self.penalty = p
        self.temperature = t
        self.name = "Activation Matrix"
    def genRandomValues(self):
        for row in self.data:
            i = 0
            while i < len(row):
                r = random.random()
                if (r < 0.6):
                    row[i] = 1
                else :
                    row[i] = 0
                i += 1

    def decay(self):
        self.temperature = self.temperature * 0.95

    def calcDeltaC(self, row, column, matrix):
        activation = self.data[row][column]
        step = (1.0 - 2.0 * activation)
        #sums the column before, the column after, the column we are in
            and the row we are in
        sum = 0
        if (column == 0):
            for i in range(len(matrix.data)):
                #sum += -1 * self.data[i][len(self.data[i]) - 1] *
                    matrix.data[row][i]
                sum += -1 * self.data[i][column + 1] * matrix.data[row][i]
        elif (column == (len(matrix.data) - 1)):
            for i in range(len(matrix.data)):
                sum += -1 * self.data[i][column - 1] * matrix.data[row][i]
                #sum += -1 * self.data[i][0] * matrix.data[row][i]
        else:
            for i in range(len(matrix.data)):
                sum += -1 * self.data[i][column - 1] * matrix.data[row][i]
                sum += -1 * self.data[i][column + 1] * matrix.data[row][i]
        #calculate the same row and column
        for i in range (len(matrix.data)):
            if (i != row ):
                sum += -1 * self.data[i][column] * self.penalty
            else:
                pass
            if (i != column ):
                sum += -1 * self.data[row][i] * self.penalty
            else:
                pass

        step = step * ((self.bonus * activation) + sum)
        return step

    def probAcceptance(self, deltaC):
        return (1.0 / (1.0 + math.exp(-1.0 *
             (deltaC/float(self.temperature)))))

class MatrixPacket(BasicMatrix):

    def copyMatrix(self, activationMatrix):
        self.name = "Activation(p) Matrix"
        for i in range(len(activationMatrix.data)):
            self.data[i] = activationMatrix.data[i][:]

    def calculateFinalDistance(self, disMatrix):
        self.distance = 0
        calcDis = 0
        for i in range(len(self.data)):
            for j in range(len(self.data[i])): #iterate over every row
                if ((j == len(self.data[i]) - 1) and (self.data[j][i] ==
                     0)):
                    self.patchIt(i)
                    self.distance = 10000000
                    return False # we did not successfully calculate the
                         distance
                if (self.data[j][i] != 1):
                    continue
                else :
                    calcDis += 1
                    if (calcDis == 1):
                        temp1 = j
                        start = j
                    else :
                        self.distance += disMatrix.data[temp1][j]
                        temp1 = j
                        calcDis = 1
                    break
        self.distance += disMatrix.data[j][start]
        return True

    def patchIt(self, column):
        for i in range(len(self.data)):
            for j in range(len(self.data[i])):
                if (self.data[i][j] == 1):
                    break
                elif ((j == len(self.data[i]) - 1) and (self.data[i][j]
                     == 0)):
                    self.data[i][column] = 1
                    return


def findSD(locMatrix):
    actMatrix = ActivationMatrix()
    actMatrix.genRandomValues()
    actMatrix.init(1600, 1750, 550)
    #print ("T",actMatrix.temperature, actMatrix)
    #generate every posibility
    # @todo: better explain what I am doing here
    positions =
         [[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[]]
    while (actMatrix.temperature > 50):
        for rows in positions:
            for i in range(23):
                rows.append(i)
        done = False

        while not done:
            #find which random element in the matrix we are calculating
            while True:
                row = random.randrange(0, len(positions))
                #checks if this row is not empty
                if (len(positions[row]) > 0):
                    break
                #if it is we try again
            # we know there is an element in this row
            element = random.randrange(0, len(positions[row]))
            #we have to extract the number not the position due to the
                 fact that we delete elements
            column = positions[row][element]
            #removes the appropriate element
            if (element + 1 >= len (positions[row]) ): #checks if element
                 was the last column
                positions[row] = positions[row][:element]
            else:
                positions[row] = positions[row][:element] +
                     positions[row][element+1:]

            deltaC = actMatrix.calcDeltaC(row, column, locMatrix)
            acceptance = actMatrix.probAcceptance(deltaC)
            r = random.random()
            if ( r < acceptance ):
                actMatrix.data[row][column] = 1 -
                     actMatrix.data[row][column]
            else:
                pass
            #print ("deltaC: ",deltaC, " ", "acceptance: ", acceptance, "
                ", "r: ", r, "\n",actMatrix)
            #print ("x & y coords: ", row, column)
            #check if we iterated over the entire matrix
            zeroRows = 0
            for i in range(len(positions)):
                if ( len(positions[i]) > 0):
                    done = False
                    break
                else :
                    zeroRows += 1
                if (zeroRows == len(positions)):
                    done = True
        actMatrix.decay()

    packet = MatrixPacket()
    packet.copyMatrix(actMatrix)
    if (not packet.calculateFinalDistance(locMatrix)):
        packet.calculateFinalDistance(locMatrix)
        packet.calculateFinalDistance(locMatrix)
    return packet

def main():
    fileName = "MatrixData"
    locMatrix = Matrix(fileName)
    print (locMatrix)
    random.seed(42)
    minPacket = findSD(locMatrix)
    for i in range(10000):
        tempPacket = findSD(locMatrix)
        if (tempPacket.distance < minPacket.distance):
            minPacket = tempPacket
    print (minPacket, minPacket.distance)

if __name__ == '__main__':
    main()
```