Compito 2.1 di Polizzi Lucrezia, matricola: 4935449. Il testo dell'esercizio richiedeva di implementare l'algoritmo di taglio minimo di tipo Montecarlo e usarlo sopra il c.d. "grafo di Fritsch", un grafo a 9 vertici e 21 archi. Bisognava applicare il MCMinCut 10^5 volte e calcolare la frequenza empirica "p" con la quale si otteneva il taglio minimo, con questo valore si doveva poi calcolare il numero di run necessari per ottenere il taglio minimo con una probabilità del 99.9%.

Chiamando MCMinCut sul grafo si otteneva un taglio minimo pari a 4 ed una frequenza empirica che si aggira mediamente intorno alle 22698 volte.

La probabilità generale di ottenere il taglio minimo si aggira intorno al 23%, mentre il numero necessario di run per ottenere il taglio minimo ad una probabilità del 99.9% è circa 27.

Di seguito ecco il codice:

```
import random
import math
import numpy

# nodo di una lista
class Node:
	# costruttore
	def __init__(self, data = None, next=None):
		self.data = data
		self.next = next

# linked list con un singolo nodo
class LinkedList:
	def __init__(self):
		self.head = None

	# inserimento in una linked list
	def insert(self, data):
		newNode = Node(data)
		if(self.head):
			current = self.head
			while(current.next):
				current = current.next
			current.next = newNode
		else:
			self.head = newNode

	# cancello il primo elemento che ha la chiave passata per
argomento
	def deleteNode(self, key):
		# salvo la testa in una variabile temporanea
		temp = self.head
		#se i nodo in testa non è nullo e rispetta la chiave lo
```

```
cancello
        if (temp != None):
            if (temp.data == key):
                self.head = temp.next
                temp = None
                return
        #se non è il nodo in testa cerco il nodo da cancellare e mi
salvo il nodo
        #precedente perché devo aggiustare i puntatori
        while(temp != None):
            if temp.data == key:
                break
            prev = temp
            temp = temp.next

        if(temp == None):     # se non ci sono nodi che rispettano
la ricerca ritorno
            return

        prev.next = temp.next      # Stacco il nodo dalla linked
list
        temp = None


def creaFritsch():# grafo di Fritsch
    Fritsch = {}

    # nodo A
    NodoA = LinkedList()
    NodoA.insert('B')
    NodoA.insert('C')
    NodoA.insert('D')
    NodoA.insert('H')
    NodoA.insert('I')
    Fritsch['A'] = NodoA

    # nodo B
    NodoB = LinkedList()
    NodoB.insert('A')
    NodoB.insert('D')
    NodoB.insert('E')
    NodoB.insert('H')
    Fritsch['B'] = NodoB

    # nodo C
    NodoC = LinkedList()
    NodoC.insert('A')
    NodoC.insert('D')
    NodoC.insert('F')
    NodoC.insert('I')
```

```python
Fritsch['C'] = NodoC

# nodo D
NodoD = LinkedList()
NodoD.insert('A')
NodoD.insert('B')
NodoD.insert('C')
NodoD.insert('E')
NodoD.insert('F')
Fritsch['D'] = NodoD

# nodo E
NodoE = LinkedList()
NodoE.insert('B')
NodoE.insert('D')
NodoE.insert('F')
NodoE.insert('H')
NodoE.insert('G')
Fritsch['E'] = NodoE

# nodo F
NodoF = LinkedList()
NodoF.insert('C')
NodoF.insert('D')
NodoF.insert('E')
NodoF.insert('G')
NodoF.insert('I')
Fritsch['F'] = NodoF

# nodo G
NodoG = LinkedList()
NodoG.insert('E')
NodoG.insert('F')
NodoG.insert('H')
NodoG.insert('I')
Fritsch['G'] = NodoG

# nodo H
NodoH = LinkedList()
NodoH.insert('A')
NodoH.insert('B')
NodoH.insert('E')
NodoH.insert('G')
NodoH.insert('I')
Fritsch['H'] = NodoH

# nodo I
NodoI = LinkedList()
NodoI.insert('A')
NodoI.insert('C')
```

```python
    NodoI.insert('F')
    NodoI.insert('G')
    NodoI.insert('H')
    Fritsch['I'] = NodoI

    return Fritsch


# conta i nodi in un grafo
def contoNodi(grafo):
    cont = 0
    for key in grafo:
        cont = cont + 1

    return cont


# campiona un nodo dal grafo
def campionaNodo(grafo):
    n = random.randint(0, contoNodi(grafo)-1)

    count=0
    for elem in sorted(grafo.keys()):
        if n==count:
            return elem
        else:
            count+=1

def contElem(adjList):
        counter = 0
        current = adjList.head
        while(current.next):
            counter = counter + 1
            current = current.next

        return counter+1


# campiono un nodo adiacente ad un nodo
def campionaNodoAdj(grafo, keyNode):
    n = random.randint(0, contElem(grafo[keyNode])-1)

    current = grafo[keyNode].head

    count=0
    while(n > count):
        current = current.next
        count = count + 1
```

```
        return current.data


# collasso i due nodi in un nodo solo
# unendo le liste di adiacenza ed eliminando da esse i riferimenti ai
nodi stessi
def collassaNodo(grafo, keyA, keyB):
        adjListAB = LinkedList()

        # inserisco nella nuova lista gli elementi della adjList di A
        current = grafo[keyA].head
        while(current):
                # se non è uguale all'altro nodo
                if(current.data != keyB):
                        adjListAB.insert(current.data)
                current = current.next

        # inserisco nella nuova lista gli elementi della adjList di B
        current = grafo[keyB].head
        while(current):
                if(current.data != keyA):
                        adjListAB.insert(current.data)
                current = current.next


        # sostituisco in tutte le altre liste di adiacenza i nodi A e B
con il nodo AB
        for elem in sorted(grafo.keys()):
                current = grafo[elem].head
                while(current):
                        if(current.data == keyA):
                                grafo[elem].deleteNode(keyA) # levalo dalla
lista di adiacenza
                                grafo[elem].insert(keyA + keyB) # inserisci il
nuovo

                        if(current.data == keyB):
                                grafo[elem].deleteNode(keyB) # levalo dalla
lista di adiacenza
                                grafo[elem].insert(keyA + keyB) # inserisci il
nuovo

                        current = current.next

        # rimuovo i due nodi dalla lista
        grafo.pop(keyA)
        grafo.pop(keyB)

        # aggiungo il nodo derivante dall'unione
```

```python
    grafo[keyA + keyB] = adjListAB

    # ne ritorno la chiave
    return (keyA + keyB)


# ottiene un taglio minimo
def getMinCut(grafo):
    # fino a quando il numero di nodi è maggiore di due
    while(contoNodi(grafo) > 2):
        # campiono un nodo
        keyRandNode = campionaNodo(grafo)
        # campiono un nodo adiacente ad esso
        keyRandAdjNode = campionaNodoAdj(grafo, keyRandNode)
        # collasso i due nodi in un nodo solo
        # unendo le liste di adiacenza ed eliminando da esse i
riferimenti ai nodi stessi
        keyNewNode = collassaNodo(grafo, keyRandNode,
keyRandAdjNode)

    # conto e ritorno quanti archi hanno i due nodi
    return contElem(grafo[keyNewNode])

if __name__ == "__main__":
     # dichiara un dizionario (taglioMin, frequenza)
     tagliDiz = {}

     # applica MCMinCut 10^5 volte
     R = 100000
     for i in range(R) :

         # Genera il grafo di Fritsch
          Fritsch = creaFritsch()
         # calcolo un taglio minimo
          taglioMin = getMinCut(Fritsch)

         # lo metto nel dizionario (taglioMin, frequenza)
          if taglioMin not in tagliDiz: # se non è già nel
dizionario lo aggiungo
               tagliDiz[taglioMin] = 1
          else:# altrimenti aumento di uno le volte in cui ho già
avuto quel taglio
               tagliDiz[taglioMin] = tagliDiz[taglioMin] + 1

     # per stampare il taglio minimo ordino il dizionario
   # usando key = lambda x: x[0], così faccio in modo che si ordini in
base a x[0],
     # cioè il taglio
     p = min(tagliDiz.items(), key=lambda x: x[0])
```

```
      print("Taglio minimo: ",str(p[0]))
   #stampo la frequenza empirica con la quale ottengo un taglio minimo
      print("Frequenza empirica con la quale ottengo un taglio minimo:
",str(p[1]))

      #Utilizza pˆ per calcolare il numero di run R necessari per
      #ottenere il taglio minimo con una probabilita del 99.9%.
      pr= p[1]/R #probabilità di ottenere il taglio minimo
      print("Probabilità di ottenere il taglio minimo: ", pr*100, "
%")

      numR = -7/(numpy.log(1-pr)) #numero di run necessari per avere
il taglio minimo con una probabilita del 99.9%.
      print("Il numero necessario di run per ottenere il taglio minimo
ad una probabilità del 99.9% sono circa: ", math.trunc(numR))
```

```
Taglio minimo:  4
Frequenza empirica con la quale ottengo un taglio minimo:  22646
Probabilità di ottenere il taglio minimo:  22.646  %
Il numero necessario di run per ottenere il taglio minimo ad una
probabilità del 99.9% sono circa:  27
```