**Pontificia Universidad Católica Madre y Maestra**

**(PUCMM)**

# File:EscudoPucmm.gif

**Nombre:**

Máximo Rodríguez

**Matrícula:**

**(**2010-2172**)**

**Nombres de la práctica:**

T6 – Greedy

**Profesor:**

Juan R. Núñez P.

**Introducción**

Un algoritmo voraz (también conocido como greedy en Ingles, devorador o goloso) es aquel que, para resolver un determinado problema, sigue una [heurística](https://es.wikipedia.org/wiki/Heur%C3%ADstica) consistente en elegir la opción óptima en cada paso local con la esperanza de llegar a una solución general óptima. Este esquema algorítmico es el que menos dificultades plantea a la hora de diseñar y comprobar su funcionamiento. Normalmente se aplica a los [problemas de optimización](https://es.wikipedia.org/wiki/Optimizaci%C3%B3n_(matem%C3%A1tica)).

Una aproximación **voraz** consiste en que cada elemento a considerar se evalúa una única vez, siendo descartado o seleccionado, de tal forma que si es seleccionado forma parte de la solución, y si es descartado, no forma parte de la solución ni volverá a ser considerado para la misma. Una forma de ver los algoritmos voraces es considerar la estrategia de [Vuelta atrás](https://es.wikibooks.org/wiki/Algoritmia/Vuelta_atr%C3%A1s), en la cual se vuelve recursivamente a decisiones anteriormente tomadas para variar la elección entonces tomada, pero eliminando esa recursión y eligiendo la mejor opción.

El término voraz se deriva de la forma en que los datos de entrada se van tratando, realizando la elección de desechar o seleccionar un determinado elemento una sola vez.

Al contrario que con otros métodos algorítmicos, no siempre es posible dar una solución a un problema empleando un algoritmo voraz. No todos los problemas son resolubles con algoritmos voraces.

**Procedimiento**

1-En este problema de programación y al siguiente codificaras los algoritmos voraces para minimizar el peso de tiempos de ejecución. Usaremos el archivo jobs.txt. Este archivo describe un conjunto de trabajos con pesos y longitudes positivas e integrales. Tiene el formato

[number\_of\_jobs]

[job\_1\_weight] [job\_1\_length]

[job\_2\_weight] [job\_2\_length]

...

Por ejemplo, la tercera línea del archivo es "74 59", lo que indica que el segundo trabajo tiene un peso 74 y la longitud 59. Usted no debe asumir que pesos de las aristas o longitudes son diferentes. Su tarea en este problema es ejecutar el algoritmo voraz que planifica los trabajos en orden decreciente de la diferencia (peso - longitud). Recordemos que este algoritmo no siempre es óptimo.

IMPORTANTE: si dos trabajos tienen la misma diferencia (peso - longitud), debe programar el trabajo con un mayor peso en primer lugar. Ten cuidado: si rompe empates de una manera diferente, es probable que obtenga la respuesta equivocada. Debe reportar la suma de tiempos de finalización en su reporte

CONSEJO: Para estar seguros, probar algunos pequeños casos de prueba para depurar el algoritmo

**Código**

def weighted\_complete\_time(jobs):

sorted\_jobs = ratio\_sort(jobs)

complete\_time = time\_to\_complete(sorted\_jobs)

total\_time = sum([i[0] \* i[1] for i in complete\_time])

return total\_time

def ratio\_sort(jobs):

ratio\_jobs = [i + [float(i[0]) / i[1]] for i in jobs]

sorted\_jobs = sorted(ratio\_jobs, key=lambda job: job[-1], reverse=True)

return sorted\_jobs

def time\_to\_complete(jobs):

current\_complete\_time = 0

for job in jobs:

job[1] += current\_complete\_time

current\_complete\_time = job[1]

return jobs

def main():

jobs = []

with open('jobs.txt') as file\_in:

next(file\_in)

for line in file\_in:

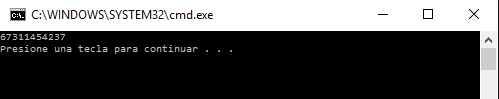
jobs.append(map(int, line.strip().split(' ')))

return weighted\_complete\_time(jobs)

if \_\_name\_\_ == '\_\_main\_\_':

print main()

**Salida**

****

2 - Para este problema, utilizar los mismos datos establecidos como en el problema anterior. Su tarea ahora es ejecutar el algoritmo voraz que planifica los trabajos (de forma óptima) tomando en cuenta la relación (peso / longitud) en orden decreciente. En este algoritmo, no importa cómo se rompan los empates. En el reporte debes indicar la suma de tiempos de finalización ponderados del planeamiento resultante --- un número entero positivo.

**Código**

def weighted\_complete\_time(jobs):

sorted\_jobs = diff\_sort(jobs)

complete\_time = time\_to\_complete(sorted\_jobs)

total\_time = sum([i[0] \* i[1] for i in complete\_time])

return total\_time

def diff\_sort(jobs):

diff\_jobs = [i + [i[0] - i[1]] for i in jobs]

sorted\_jobs = sorted(diff\_jobs, key=lambda job: (job[-1], job[0]), reverse=True)

return sorted\_jobs

def time\_to\_complete(jobs):

current\_complete\_time = 0

for job in jobs:

job[1] += current\_complete\_time

current\_complete\_time = job[1]

return jobs

def main():

jobs = []

with open('jobs.txt') as file\_in:

next(file\_in) #start reading from line 2

for line in file\_in:

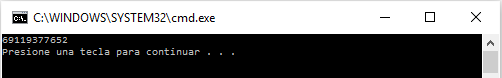
jobs.append(map(int, line.strip().split(' ')))

return weighted\_complete\_time(jobs)

if \_\_name\_\_ == '\_\_main\_\_':

print main()

**Salida**

****

Arbol de expansion minima

En este problema de programación usted debe codificar el algoritmo de árbol de expansión mínima de Prim. Usaremos el archivo de texto edges.txt. Este archivo describe un grafo no dirigido con longitudes de aristas valores enteros. Tiene el formato

[numero\_de\_nodos] [numero\_de\_aristas]

[un\_nodo\_de\_arista\_1] [otro\_nodo\_de\_arista\_1] [arista\_1\_costo]

[un\_nodo\_de\_arista\_2] [otro\_nodo\_de\_arista\_2] [arista\_2\_costo]

Por ejemplo, la tercera línea del archivo es "2 3 -8874", que indica que hay un borde de conexión de vértices # 2 y # 3 vértices con peso -8874. Usted no debe asumir que los costos de borde son positivos ni debe asumir que son distintos.

Su tarea es ejecutar el algoritmo de árbol de expansión mínima de Prim en este gráfo. Usted debe informar, en su reporte, el costo total de un árbol de expansión mínimo --- un entero, que puede o no puede ser negativo.

**Código**

**File----dijkstra.py---**

import heapq

class PriorityQueue(object):

def \_\_init\_\_(self, heap=[]):

heapq.heapify(heap)

self.heap = heap

self.entry\_finder = dict({i[-1]: i for i in heap})

self.REMOVED = '<remove\_marker>'

def insert(self, node, priority=0):

if node in self.entry\_finder:

self.delete(node)

entry = [priority, node]

self.entry\_finder[node] = entry

heapq.heappush(self.heap, entry)

def delete(self, node):

entry = self.entry\_finder.pop(node)

entry[-1] = self.REMOVED

return entry[0]

def pop(self):

while self.heap:

priority, node = heapq.heappop(self.heap)

if node is not self.REMOVED:

del self.entry\_finder[node]

return priority, node

raise KeyError('pop from an empty priority queue')

def dijkstra(source, pq, edges):

size = len(pq.heap) + 1

processed = [source]

uncharted = set([i[1] for i in pq.heap])

shortest\_path = {}

shortest\_path[source] = 0

while size > len(processed):

min\_dist, new\_node = pq.pop()

processed.append(new\_node)

uncharted.remove(new\_node)

shortest\_path[new\_node] = min\_dist

for head, edge\_dist in edges[new\_node]:

if head in uncharted:

old\_dist = pq.delete(head)

new\_dist = min(old\_dist, min\_dist + edge\_dist)

pq.insert(head, new\_dist)

return shortest\_path

**File ---mst.py---**

from dijkstra import PriorityQueue as PQ

from collections import defaultdict

def Prim(source, pq, edges):

processed = [source]

mst = []

uncharted = set([i[1] for i in pq.heap])

while len(uncharted) != 0:

cheapest\_edge\_dist, new\_node = pq.pop()

processed.append(new\_node)

mst.append(cheapest\_edge\_dist)

uncharted.remove(new\_node)

for head, edge\_dist in edges[new\_node]:

if head in uncharted:

old\_dist = pq.delete(head)

new\_dist = min(old\_dist, edge\_dist)

pq.insert(head, new\_dist)

return mst

def main():

edges = defaultdict(list)

heap = []

source = 1

BIG = 1000000000

with open('edges.txt') as file\_in:

#with open('primcase.txt') as file\_in:

for i, line in enumerate(file\_in):

x = line.strip().split()

if i == 0:

number\_of\_nodes = int(x[0])

else:

head, tail, edge\_dist = int(x[0]), int(x[1]), int(x[2])

edges[head].append((tail, edge\_dist))

edges[tail].append((head, edge\_dist))

source\_edges = dict(edges[source])

for node in range(2, number\_of\_nodes + 1):

if node in source\_edges.keys():

heap.append([source\_edges[node], node])

else:

heap.append([BIG, node])

pq = PQ(heap)

mst\_edges = Prim(1, pq, edges)

return sum(mst\_edges)

if \_\_name\_\_ == '\_\_main\_\_':

print main()

**Salida:**

****