Proyecto Análisis de Algoritmos

Javier Esteban Sandoval Albarracín, Stephanie Dominguez, etc.
Ingeniería de Sistemas,
Pontificia Universidad Javeriana
Sandoval.javier@javeriana.edu.co, s.dominguez@javeriana.edu.co

***Resumen*- Este documento presenta el análisis primario respecto a los problemas presentados en el enunciado del proyecto semestral para la materia Análisis de Algoritmos.**

***Palabras Clave*- MST, Grafo, Prim, algoritmo, Kruskal, Dijkstra, Grafo, árbol, recurrencia, divide and conquer, arreglo, cola de prioridad.**

I. Introducción

El análisis de algoritmos ha sido fundamental en el desarrollo de la computación moderna, desde el nacimiento de las matemáticas existen maneras de encontrar soluciones a problemas triviales y no tan triviales de la sociedad. Un algoritmo representa una serie de pasos con una entrada definida y una salida esperada, para esta definición nuestra solución en esta primera entrega representa una caja negra con la explicación detallada del problema y sus posibles soluciones.

II. Glosario

MST: Minimun Expanding Tree(Árbol de Expansión mínima)

LP: Linear Programing

III. MST

En esta sección se realizará la descripción detallada de los problemas planteados, de forma que el lector comprenda las instancias reales y las situaciones posibles que se pueden presentar.

Resolver el árbol de expansión mínima: El árbol de expansión mínima según el profesor Erik Damin phD de MIT define matemáticamente un MST como:

$$Dado\ un\ Grafo\colon G(v,e) y\ una\ arista\ e \rightarrow W\colon e \in R\ :\ \sum_{e \in T} w(e)$$

el proyecto plantea una variación a este clásico de la programación es cada arista de nuestro grafo es coloreada azul o roja donde el número de aristas pertenecientes a cada color se rigen de la siguiente forma:

k aristas azules

$n - k - 1$ *aristas rojas*

Entrada al algoritmo: Para el algoritmo debe existir un archivo describiendo el grafo en texto plano con el número de aristas en la primera línea y luego una relación de tipo:

idNodo a…n(id nodos adyacentes) Azul-Rojo

Salida: Nodos Azules: k
Nodos Rojos: n – k – 1

A. Solución

a. Algoritmo Voraz

Un algoritmo voraz es aquel que permite una substracción de las soluciones del árbol de manera tal que selecciona la solución que más se acomoda a un parámetro fijado.
(Divide and Conquer)

Algoritmo de kruskal: Dado un grafo no dirigido el objetivo principal es unir todos los vértices menos 1 se cree un MST. Los pasos para el algoritmo son los siguientes:

a. Ordenar todas las aristas con sus pesos.
b. Escoger la arista con el peso más pequeño, verificar si hay un ciclo, es decir que si al escoger esta arista no se cumple con el principio básico que forma un árbol.
c. Repetir los pasos 1 y 2 hasta que se en el árbol se forme.
d. La complejidad del algoritmo de Kruskal se define como:
e. Mantener todos los componentes conectados y usar una estructura que se llama unir y encontrar.
f. Se empieza con un set vacío.
h. Ordenar por pesos de E
i. Para se tiene en orden incremental

j. Si no encontramos el set que une u y v lo unimos en nuestro árbol

k. La complejidad de este algoritmo:

l. O(sort (E)) + E (v) –v //mirar si se pone

m. LA correctitud de este algoritmo se define por la siguiente invariente: El sub árbol que se escoge pertenece a un árbol de expansión minima T'

n. Asumiendo que t

o. T -> t' t U {u}

Y la segunda parte del algoritmo es el Greedy Choice Property que toma opciones de solución local óptima de forma que estas me conduzcan a la solución global óptima.

Se sigue una Heurística aplicada a problemas de optimización, simples de forma que en la solución se me represente un árbol conectado aciclico. *b. Algoritmo de Prim*

Debemos escoger un nodo S que será el punto de partida para el camino, existe paralelamente una cola de prioridad que almacena los nodos del grafo.

Pasos:

1. Invariante: $v.key = \min\{w(u, v) | u \in S\}$
2. Inicializar Todo{Cola, el grafo, invariante}
3. For $v \in V\{s\}$ $v.key =$ ∞ *donde v representa los vértices.*
4. Hasta que la cola quede vacía:
   U: extrac_min(cola) se agrega u a subgrafo S
5. Sigue un pseudocódigo que se implementará en la segunda entrega

c. Ejemplo Prim

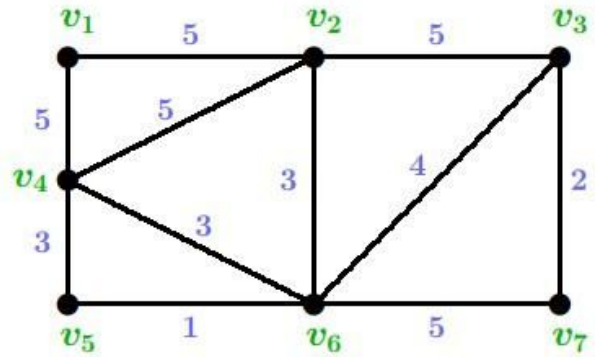
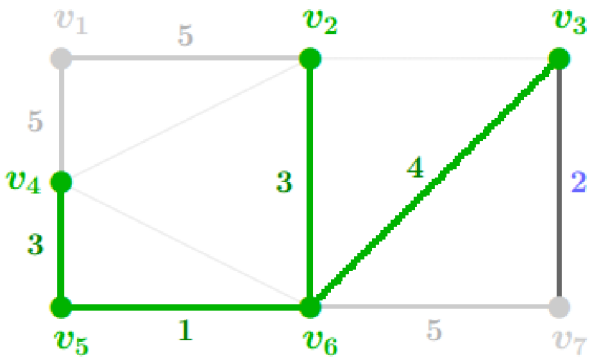


Figura 1. Grafo



$W = \{ v_2, v_6, v_5, v_4, v_3$

$B = \{\{2,6\}\ \{5,6\}\ \{4,5\}\ \{3,6\}\ \}$

Figura 2. Paso Intermedio Algoritmo Prim

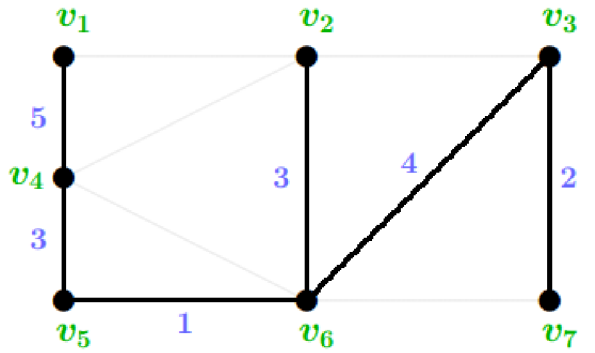


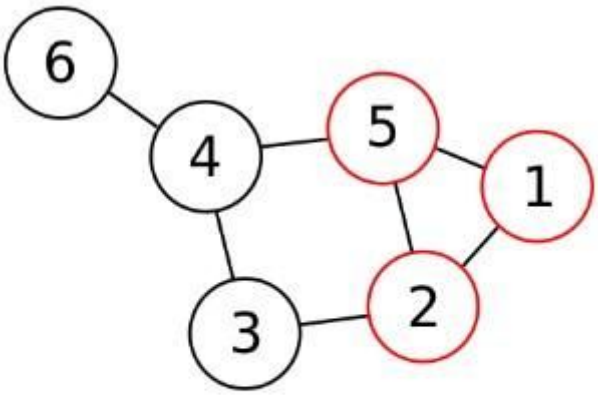
Figura 3. Solución final Algoritmo de Prim, donde Azules = 6, y Rojas = 5, donde el árbol son las aristas azules en este caso señaladas de Negro.

IV. Distancia de Hamming

La distancia de Hamming en palabras sencillas es el número de elementos que varían de un arreglo a otro tal que los dos arreglos tengan un mismo tamaño n. Matemáticamente se define como una función *dist(u,v)* entre vectores U y V de n elementos tal que de los elementos k_i para los dos vectores sean diferentes para i = j. Nuestro algoritmo debe hallar el MWC del grafo de haming tal que

A(n, d) = max |{S ⊂ {0, 1} n | dist(u, v) ≥ d for all distinct u, v ∈ S}| y A(n, d) = H(n, d) donde n es el tamaño del grafo y d es la distancia de Hamming entre dos grafos. Se debe diseñar un algoritmo que me permita encontrar el máximo de arreglos que se pueden crear con dichas condiciones, es decir que difieran en esos *d* números. Las condiciones es que los vectores sean te tipo binario.

Entrada: Se generará un grafo de Hamming a partir de n = tamaño del vector, y m aristas del grafo.
Salida: número de subgrafos generados que sean completos dentro del grafo de Hamming. A. Ejemplo:



The graph shown has one maximum clique, the triangle {1,2,5}, and four more maximal cliques, the pairs {2,3}, {3,4}, {4,5}, and {4,6}.

Figura 4. Problema de Cliques en Grafo[2]

Solución:

Salida: 1 Máximo clique que interconecta 3 nodos

Un Clique es el número máximo de componentes totalmente conectados dentro de un grafo, como se ilustra en la figura 4 el máximo número de componentes es 3 (sub-grafo completo).

Algoritmo Propuesto:

Nuestro algoritmo consta de listar todos los Cliques Máximos producidos por el grafo con una complejidad promedio de 2^n ya que siempre se recorren todos los vértices por lo tanto el número de vértices es igual al número de operaciones.

Algoritmo propuesto 2:

Para mejorar este algoritmo como segunda opción de solución podríamos implementar un algoritmo paralelo que recorra el grafo y guarde los que ya recorrió de manera que se evite recorrerlos varias veces. Dicha complejidad sería de la forma O(V log(v))

Algoritmo de Dijkstra: Resuelve el problema de hallar el camino de longitud mínima entre dos grafos, de aquí conocemos las complejidades aproximadas de nuestra propuesta.

V. LP

Problema de programación lineal dado para un caso de optimización de recursos, se abordará de acuerdo a lo visto en las soluciones con algoritmos de simplex y en dado caso de un método gráfico para pocas tiendas que puedan representarse como ecuaciones restrictivas en la solución.

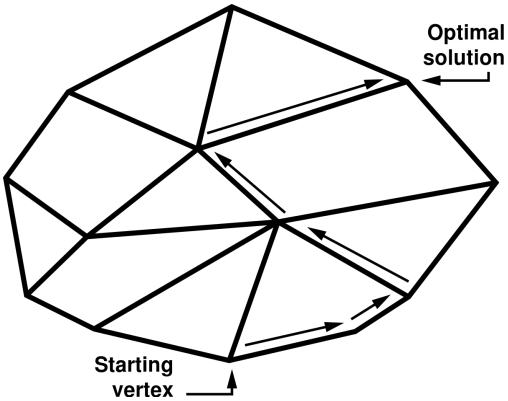


Figura 5.Solución Simplex

Este algoritmo plantea una matriz donde sus columnas y filas son las restricciones y variables de holgura pertinentes para resolver cada problema, dichos campos varían su valor según se haga una relación entre ellos durante cada iteración, el mayor valor va saliendo de las restricciones de manera que al final quede una solución óptima.

VI. SOLUCIONES

a. LP

```
#Algoritmo Dijkstra: Referenciado de https://gist.github.com/shi
def dijkstra(matrix,m,n,k):

    cost=[[0 for x in range(m)] for x in range(1)]
    offsets = []
    offsets.append(k)
    elepos=0
    for j in range(m):
        cost[0][j]=matrix[k][j]
    mini=999
    for x in range (m-1):
        mini=999
        for j in range (m):
                if cost[0][j]<=mini and j not in offsets:
                        mini=cost[0][j]
                        elepos=j
        offsets.append(elepos)
        for j in range (m):
            if cost[0][j] >cost[0][elepos]+matrix[elepos][j]:
                cost[0][j]=cost[0][elepos]+matrix[elepos][j]
    return cost
```

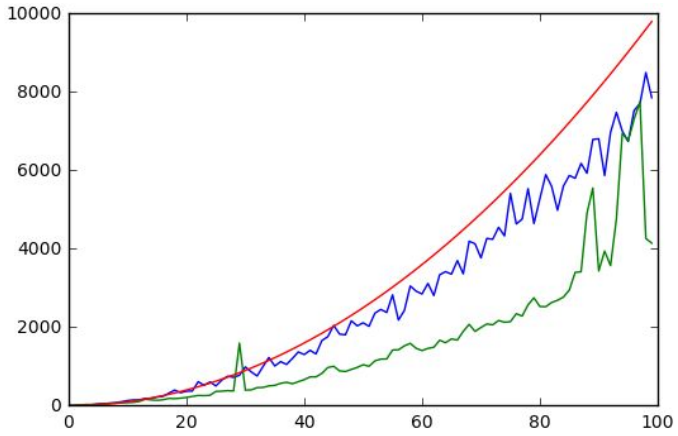
```
def variasTFijas(numtiendas,promedios):
    tiendas =[]
    prom = promedios[:];
    n = len(promedios)
    for x in range(numtiendas):
        tiendas.append(x)


    for i in range(0,numtiendas):
        index = 0;
        a = False
        for j in range (0,n):
            if promedios[j] == min(prom):
                index = j;
                if index in tiendas:
                    a = True
        if a==False:
            tiendas[i]=index;
            prom.remove(min(prom))

    return tiendas
```

En este algoritmo de programación lineal primero se realiza un recorrido en dijkstra para conocer todos los caminos más cortos entre los nodos del grafo, este algoritmo tiene un costo en tiempo de ejecución de (n^2) al ser un problema de la clase P su solución se da en tiempo polinomial, para cada punto hay que realizar el algoritmo de dijkstra, por lo que la complejidad de nuestra solución pasa a ser de N*(Dijkstra_Complex) = N*(n^2) y que asintóticamente se convierte en O(n^3)

```
('Debera colocar la tienda en el nodo:', 0)
('Con un Costo de:', 68)
[5, 5, 10, 6]
```

```
['A', '0', '6', '20', '2']
['B', '6', '0', '9', '19']
['C', '20', '9', '0', '18']
['D', '2', '19', '18', '0']
[[0, 6, 20, 2], [6, 0, 9, 19], [20, 9, 0, 18], [2, 19, 18, 0]]
('Debera colocar la tienda en el nodo:', [0, 1])
('Con un Costo de:', 6)
[5, 5, 10, 6]
```

Como resultados se pueden observar tiempos de ejecución acorde al criterio escogido, es decir entre más grande sea la entrada su ejecución crecerá exponencialmente según los datos que deban ser procesados.



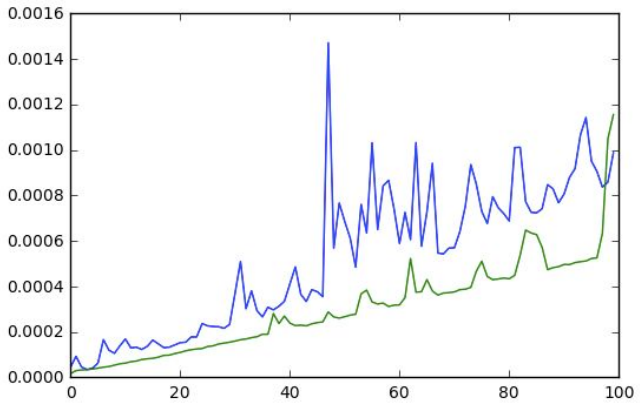
Para un grafo de 10 nodos el tiempo corre exponencialmente, el verde una sola tienda, el azul varias tienda y el rojo es el tiempo teórico de O(n^2) que realiza dijkstra.

b. MST

```
class mst(object):

    def __init__(self):
        self.parent= dict()
        self.rank= dict()
        self.minimum_spanning_tree = set()

    def make_set(self,vertice):
        self.parent[vertice] = vertice
        self.rank[vertice] = 0

    def find(self, vertice):
        if self.parent[vertice] != vertice:
            self.parent[vertice] = self.find(self.parent[vertice])
        return self.parent[vertice]

    def union(self,vertice1, vertice2):
        root1 = self.find(vertice1)
        root2 = self.find(vertice2)
        if root1 != root2:
            if self.rank[root1] > self.rank[root2]:
                self.parent[root2] = root1
            else:
                self.parent[root1] = root2
                if self.rank[root1] == self.rank[root2]: self.rank[root2] += 1

    def kruskal(self, graph, color):
        for vertice in graph['vertices']:
            self.make_set(vertice)
        cont =0
        edges = list(graph['edges'])
        edges.sort()
        for edge in edges:
            weight, vertice1, vertice2, colorEdge = edge
            if colorEdge == color:
                if self.find(vertice1) != self.find(vertice2):
                        self.union(vertice1, vertice2)
                        self.minimum_spanning_tree.add(edge)
                        cont = cont + 1
```

```python
#https://codereview.stackexchange.com/questions/86021/check-if-a-directed-graph-
def cyclic(g,save):
    path = set()
    visited = set()
    def visit(vertex):
        if vertex in visited:
            return False
        visited.add(vertex)
        path.add(vertex)
        for neighbour in g.get(vertex, ()):
            if neighbour in path or visit(neighbour):
                save.add(vertex)
                save.add(neighbour)
        path.remove(vertex)
        if len(save)>0:
            return save
        else:
            return 0
    return any(visit(v) for v in g)
```

```python
def mergMst(blue,red,contR, contB, k, n,auxMst):
    if contB >= k or contR >= n:
        return "Mst"
    dicMst = dict()
    save = set()
    auxMst= findDiference(blue.minimum_spanning_tree,red.minimum_spanning_tree)
    for aux in auxMst:
        if contB< k and contR < n:
            weight, vertice1, vertice2, colorEdge = aux
            red.union(vertice1,vertice2)
            red.minimum_spanning_tree.add(aux)
            dicMst=createAdjacentList(red.minimum_spanning_tree)
            flag=cyclic(dicMst,save)
            if flag == True:
                aux3 = set()
                aux3 = saveAsEdge(save,blue.minimum_spanning_tree)#No esta en B
                aux3=findDiference(aux3,blue.minimum_spanning_tree)
                edge= auxMethod(aux3)
                red.minimum_spanning_tree.remove(edge)
                contB= contB +1
                return mergMst(blue,red,contR, contB, k, n,auxMst )
            else:
                contR = contR + 1
                return mergMst(blue,red,contR, contB, k, n,auxMst )
```



En gráfica anterior se observa cómo nuestra predicción que están en la parte superior fueron correctas ya que está en una curva de O (V log E). Se escogieron datos acorde al criterio de selección. La gráfica azul es el tiempo con nuestro algoritmo para hallar el MST según los criterios mencionados anteriormente y el segundo es algoritmo de Kruskal solo encontrando el MST sin tener en cuenta los colores de la aristas.

VII. REFERENCIAS

[1] https://ocw.mit.edu/courses/electrical-engineering-andcomputer-science/6-046j-design-and-analysis-ofalgorithms-spring-2015/recitation-videos/

[2] http://www.wikiwand.com/en/Clique_problem

[3]MIT: https://www.youtube.com/watch?v=tKwnms5iRBU