Optimización de Flujo en Redes: Reporte 5

Mayra Cristina Berrones Reyes. 6291

Mayo 2018

Introducción

Para este reporte se abordan temas nuevos, así como algunos de los temas que ya hemos estado conociendo en reportes anteriores [2]. En cuanto a temas nuevos, nos referimos a las distancias Manhattan, que en este caso podemos definir como la distancia entre dos puntos de una malla de nodos en un camino estrictamente horizontal o vertical (dentro de las líneas de la malla), opuesto a las líneas o distancias diagonales. La distancia Manhattan es la suma simple de sus componentes verticales y horizontales, y en donde las líneas diagonales pueden ser realizadas por medio del teorema de Pitágoras, como se ha realizado en reportes anteriores. [5]

En este reporte se utilizara las distancias Manhattan para encontrar el flujo máximo que podemos pasar de un punto de inicio a un punto final de la malla, tomando estos puntos como vértices opuestos dentro de la malla de nodos. Para encontrar el flujo máximo se utilizó uno de los algoritmos que ya se han visto con anterioridad y puesto en práctica. El algoritmo de Ford-Fulkerson.

Como breve repaso, el algoritmo de Ford-Fulkerson propone buscar caminos en los que se pueda aumentar el flujo, hasta que se alcance el flujo máximo. Considérese cualquier camino dirigido del origen al destino en la red de flujos. Sea x la mínima de las capacidades de las aristas no usadas en el camino. Es posible incrementar el flujo de la red al menos en x, incrementando el flujo de las aristas del camino en dicho monto. De esta forma se obtiene el primer intento de flujo en la red. Luego debemos encontrar otro camino, incrementar el flujo en el camino, y continuar hasta que todos los caminos del origen al destino tengan al menos una arista llena (el flujo usa toda la capacidad de la arista). [4]

En este reporte se tomará como referencia el código de camino y Ford-Fulkerson que se vio en el Reporte 3, [1] en el que nos basamos de la página de la Dra. Elisa, [3] con algunas modificaciones para poder introducir nuestro nuevo grafo en forma de malla.

Modificación del grafo principal

Al igual que como se realizó en el Reporte 4 [2] se modificó la forma y la estructura de los nodos para poder realizar esta práctica. Primero re-acomodamos los nodos de manera que estuvieran representados en una distancia equidistante unos de otros, de manera cuadrangular. Para esto seleccionamos una variable k la cual representaría el número de nodos por lado que tendría nuestra malla. Viéndolo como un cubo, nuestro número de nodos final seria el de la variable k elevado al cuadrado.

```
#Extracto de código de clase Grafo()
   def puntos(self, k):
      self.k = k
```

```
with open("grafica.dat", 'w') as salida:
   for x in range(0, k):
      for y in range(0, k):
         self.nodos.append((x, y))
         if x==0 and y==k-1:
            print(x, y, 1, file = salida)
         elif x==k-1 and y==0:
            print(x, y, 6, file = salida)
         else:
            print(x, y, 7, file = salida)
```

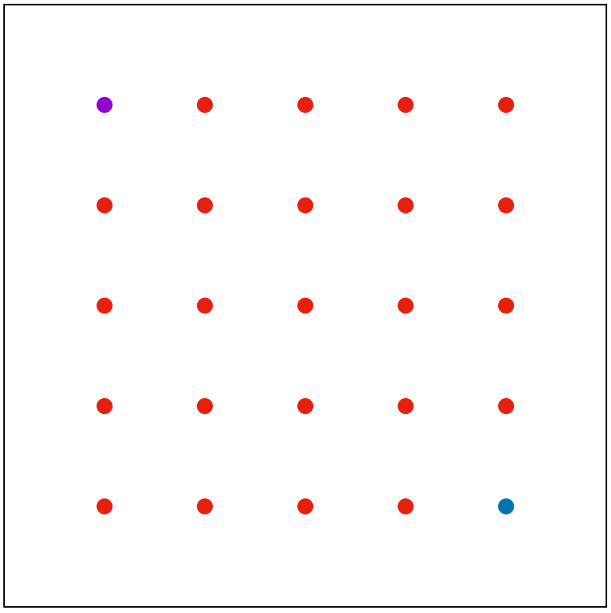


Figura 1: Nodos acomodados a manera de malla dentro del plano. La variable k en este caso es 5. Para diferenciar el nodo de inicio y el nodo final se les dieron distintos colores.

Ahora, para las conexiones se utilizaron dos tipos diferentes de caminos. Como se habló en la introducción, la base de las distancias Manhattan son aquellas que van de manera horizontal y vertical de manera estricta. En este caso usaremos la variable de l para darle valor normal a estas distancias. Pero también tenemos las distancias que aleatorias, que son las que se expresan de manera diagonal.

En este caso queremos darle un flujo exponencial, para que crucen el grafo de nodo a nodo de distinta manera al de la distancia Manhattan. Con eso podemos hacer que viaje de manera diagonal, vertical y horizontal (incluso si hace un

salto de muchos nodos). En este caso, queremos que estas líneas aleatorias sean pocas por entre el grafo, por lo que se les dará una probabilidad pequeña, y se le dan restricciones de que no deben viajar directamente desde el nodo inicial hasta el final, simplemente para que nuestro grafo nos de más información al momento de utilizarlo.

```
#Extracto de código de clase Grafo()

   def conexiones(self, l):
      self.l = l
      with open ("conexiones.dat", 'w') as salida:
         for i in self.nodos:
            for j in self.nodos:
               if i is not j:
                  if distancia (i, j) < l +1:
                     self.aristas[i, j] = self.aristas[j, i] =
                         floor(normalvariate(4, 0.5))+1

   def aleatorio(self, proba):
      self.rand = 0
      for i in self.nodos:
         for j in self.nodos:
            if i is not j:
               if j is not self.nodos[0]:
                  w = random()
                  if w < proba and (i, j) not in self.aristas:
                     self.aristas[i, j] = ceil(expovariate(.1))
                     self.rand += 1
```

En el código anterior podemos apreciar que utilizamos la herramienta de *normalvariate* y *expovariate* de la librería de *random* para darles la cantidad de flujo a cada una de las aristas.

Percolaciones

Nuevamente, tenemos un nuevo reto dentro de esta práctica. Por la forma en que esta hecho el grafo, el algoritmo de Ford-Fulkerson tarda de poco a nada de tiempo en encontrar el camino con mayor cantidad de flujo. En esta sección se juega un poco con la cantidad de aristas y nodos que tenemos en el grafo, para poder apreciar como impacta esto en el tiempo de procesamiento al momento de querer correr el algoritmo de Ford-Fulkerson en él.

A continuación se presentan ejemplos en grafos con pequeña cantidad de nodos.

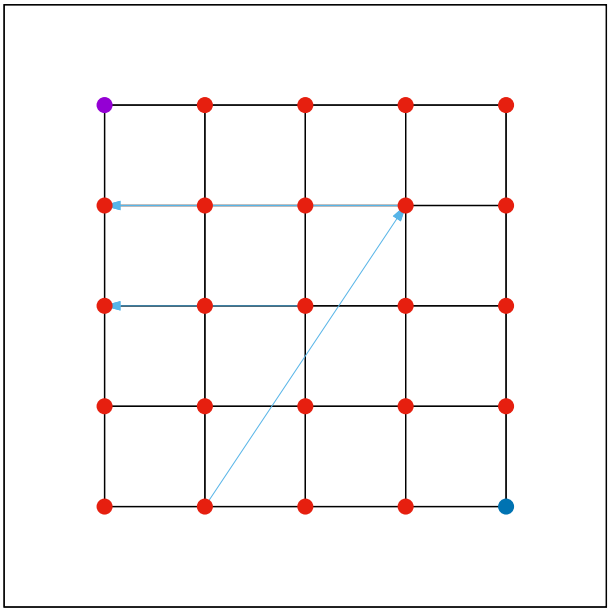


Figura 2: Conexiones de la malla. Las líneas negras representan las distancias Manhattan, mientras de las líneas azules representan las distancias exponenciales que se seleccionaron aleatoriamente. La variable l es de 1, mientras que la probabilidad que se le dio fue de 0.002.

Percolación de aristas

Para la percolación de aristas, para no demorar demasiado tiempo para ver resultados contundentes en sus grafos, se fueron quitando de más de una por una.

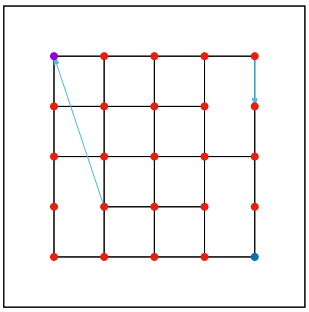
```
#Extracto de código de clase Grafo()

def perAristas(self, o):
     for i in range(0, 15):
        y = randint(1, len(self.nodos) - 1)
        num = randint(1, len(self.nodos) - 1)
        if y is not num:
           if num is not self.nodos[0]:
              if ((self.nodos[num][0],self.nodos[num][1]),
                   (self.nodos[y][0],self.nodos[y][1])) in self.aristas:
                 del
                     self.aristas[(self.nodos[num][0],self.nodos[num][1]),
                     (self.nodos[y][0],self.nodos[y][1])]
              if ((self.nodos[y][0],self.nodos[y][1]),
```

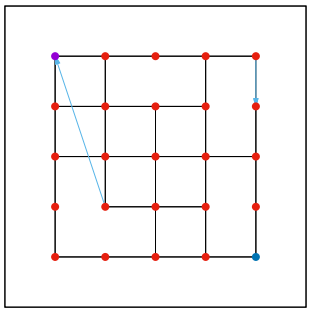
```
            (self.nodos[num][0],self.nodos[num][1])) in
            self.aristas:
        del self.aristas[(self.nodos[y][0],self.nodos[y][1]),
            (self.nodos[num][0],self.nodos[num][1])]
```

El rango que se alcanza a apreciar en las líneas de código fue más que nada porque en ambos casos de las coordenadas de nodos estamos utilizando números aleatorios, por lo que no todas las aristas que consultábamos existían, y al final solo terminaban quitando una o dos por iteración.

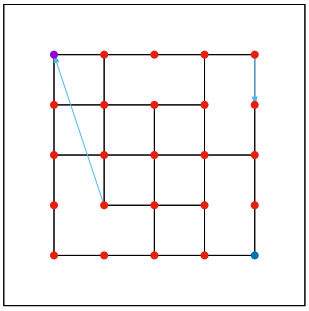
A continuación se presentan los ejemplos de grafos con los que se fueron quitando varias aristas.



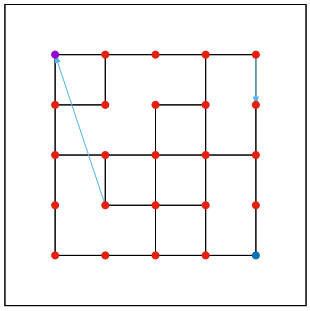
(a) Grafo p. de aristas 1



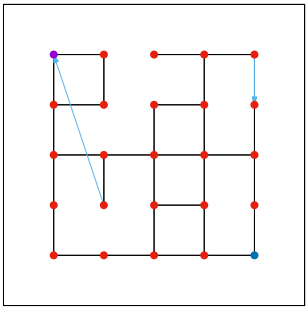
(b) Grafo p. de aristas 2



(c) Grafo p. de aristas 3



(d) Grafo p. de aristas 4



(e) Grafo p. de aristas 5

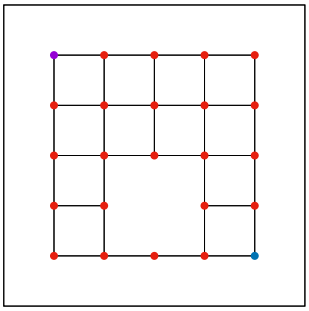
Figura 3: Grafos para percolación de aristas. Los flujos correspondientes a cada uno fueron constantes de 8, 8, 8, 8, y solo al final 4. En las figuras, p. representa percolación.

Percolación de nodos

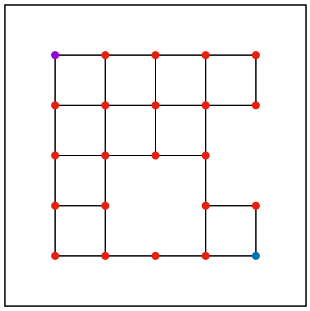
En este caso, lo que queremos ver es como afectaría quitar un nodo de la vecindad de la malla, y por consecuencia, las aristas que le acompañan. En este caso nuestra teoría es que va a ser más tardado en estos casos conforme se van quitando los nodos, o que el flujo va a tender mas rápido a cero porque se le quitan muchos caminos de manera simultánea. Al igual que en el caso anterior, se muestra un ejemplo de pocos nodos y en este caso, si se van quitando de uno por uno en cada una de las iteraciones.

```
#Extracto de código de clase Grafo()
def perNodos(self,o):
     num = randint(1, len(self.nodos) - 2)
     if num != self.nodos[len(self.nodos) - self.k -1] and num !=
         self.nodos[self.k -1]:
        for y in range (len(self.nodos)):
           if ((self.nodos[num][0],self.nodos[num][1]),
                (self.nodos[y][0],self.nodos[y][1])) in self.aristas:
              del self.aristas[(self.nodos[num][0],self.nodos[num][1]),
                  (self.nodos[y][0],self.nodos[y][1])]
           if ((self.nodos[y][0],self.nodos[y][1]),
                (self.nodos[num][0],self.nodos[num][1])) in self.aristas:
              del self.aristas[(self.nodos[y][0],self.nodos[y][1]),
                  (self.nodos[num][0],self.nodos[num][1])]
        self.nodos.pop(num)
     with open("grafica"+str(o)+".dat", 'w') as salida:
        for (x,y) in self.nodos:
           if x==0 and y==self.k -1:
              print(x, y, 1, file = salida)
           elif x==self.k-1 and y==0:
              print(x, y, 6, file = salida)
           else:
              print(x, y, 7, file = salida)
```

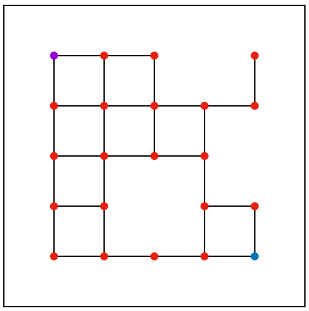
En el caso de la percolación de los nodos, se tuvo un poco mas de cuidado al momento de seleccionar cuales eran los nodos que se iban a quitar, evitando seleccionar el nodo de inicio y el nodo final, para que el recorrido pudiera seguir su trayecto por el mayor tiempo posible.



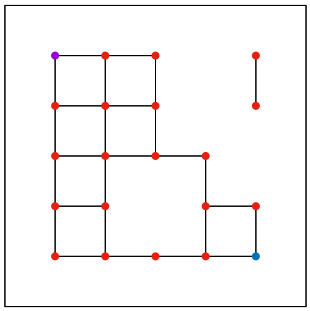
(a) Grafo p. de nodos 1



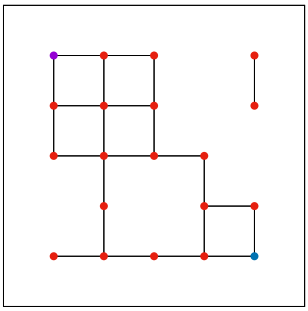
(b) Grafo p. de nodos 2



(c) Grafo p. de nodos 3



(d) Grafo p. de nodos 4



(e) Grafo p. de nodos 5

Figura 4: Grafos para percolación de nodos. Los flujos correspondientes a cada uno fueron de 10, 9, 8, 8 y 4. En las figuras, p. representa percolación.

Resultados

Para esta última fase de resultados, lo que buscamos es ver los valores en cuanto al tiempo de compilación, tanto en la forma de quitar caminos por medio de la percolación de nodos, como con la percolación de aristas.

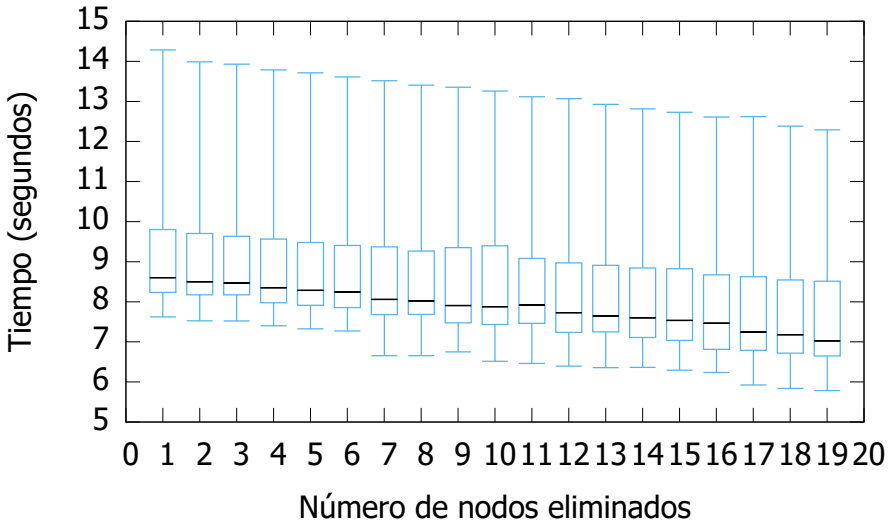


Figura 5: Caja bigote de los tiempos de percolación por nodos. En todos los casos, los grafos contaban con una k con valor de 20, una l con valor de 3, y una probabilidad de 0.003.

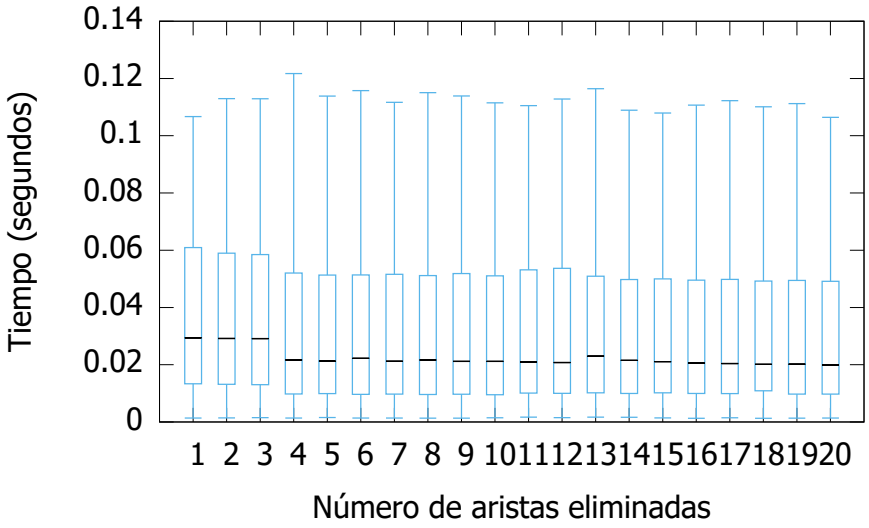


Figura 6: Caja bigote de los tiempos de percolación por nodos. En todos los casos, los grafos contaban con una k con valor de 10, una l con valor de 2, y una probabilidad de 0.003.

Conclusiones

Para este trabajo se tenía la teoría inicial de que en los valores del procesamiento de los puntos iba a tomar mucho más tiempo en encontrar el nodo final, pero por lo que se puede apreciar en las ultimas tablas, el tiempo es más grande en las de las aristas, porque el algoritmo de Ford-Fulkerson tenía muchos más caminos que recorrer, que en las que los nodos quitaban opciones. Por eso se tomó la decisión también de que debían ser mallas más pequeñas para la parte de percolación de aristas, ya que el tiempo que se tardaba en compilar la misma cantidad de vértices que cuando se realizaba la percolación a los nodos era bastante alto.

Lo único es que sí se probó correcto fue que al momento de quitar los nodos era mas factible que el algoritmo no encontrara un camino completo y nos arrojara un 0, ya que los nodos alrededor del inicio o el final desaparecían y bloqueaban el camino con mas rapidez.

Referencias

[1] Berrones Reyes, Mayra Cristina *Reporte3. Marzo 2018*

[2] Berrones Reyes, Mayra Cristina *Reporte4. Marzo 2018*

[3] Schaeffer, Elisa *https://elisa.dyndns-web.com/teaching/mat/discretas/md.html* Matemáticas Discretas. Grafos y árboles. **Consultado en Abril 2018**

[4] Algoritmo Ford-Fulkerson *Flujos Máximos. https://flujomaximo.wikispaces.com* **Consultado Mayo 2018**

[5] Distancias Manhattan *Wikitionary. https://en.wiktionary.org/wikiManhattan.distance.* **Consultado Mayo 2018**