Tarea 2: Algoritmo de caminos mínimos

Profesores: Pablo Barceló
Gonzalo Navarro
Auxiliares: Matilde Rivas
Bernardo Subercaseaux

1 Introducción

El objetivo de esta tarea es implementar y evaluar en la práctica diferentes variantes del algoritmo de Dijkstra para caminos mínimos, utilizando arreglos, heaps clásicos y heaps de Fibonacci.

Se espera que se *implementen* los algoritmos, *realicen* los experimentos correspondientes y entreguen un *informe* que indique claramente los siguientes puntos:

1. Las *hipótesis* escogidas antes de realizar los experimentos.
2. El *diseño experimental*, incluyendo la idea general de la implementación de los algoritmos, la generación de las instancias y las medidas de rendimiento utilizadas.
3. La *presentación de los resultados* en forma de una descripción textual, tablas y/o gráficos.
4. El *análisis e interpretación* de los resultados.

2 Implementación

El algoritmo de Dijkstra es un algoritmo para la determinación del camino más corto desde un vértice origen hacia el resto de los vértices en un grafo ponderado.

Se pide que implemente tres versiones de este algoritmo, una usando un heap clásico y otra utilizando heaps de fibonacci. En ambas soluciones se debe utilizar la operación `decreaseKey`. La tercera solución utiliza arreglos simples .Las estructuras deben ser de implementación propia, es decir, no puede usar una librería externa.

2.1 Implementación Naïve

La implementación naïve no utiliza colas de prioridad, sino que arreglos. Mantiene un arreglo que guarda la distancia de cada nodo hasta el origen, otro que contiene el nodo previo en el camino mínimo para llegar desde el origen y un arreglo que indica si ya conocemos la distancia definitiva. La distancia del origen al origen es 0.

```
1  dist[origen] = 0
2  dist[resto] = infinito
3  marcado[todos] = false
4  prev[todos] = null
5  for i = 1 hasta |V| − 1 do
6  |  minDist = INF
7  |  minNodo = -1 // no importa este valor
8  |  //buscar el mínimo
9  |  for  j = 1 hasta |V| do
10 |  |  if ¬marcado[j] ∧ dist[j] < minDist then
11 |  |  |  minDist = dist[j]
12 |  |  |  minNodo = j
13 |  |  end
14 |  end
15 |  u = minNodo
16 |  marcado[u] = true
17 |  //actualizar distancias
18 |  for  vecino v de u do
19 |  |  if  dist[v] > dist[u] + w(u, v) then
20 |  |  |  dist[v] = dist[u] + w(u,v)
21 |  |  |  prev[v] = u
22 |  |  end
23 |  end
24 end
```

2.2 Implementaciones con Cola de Prioridad

A continuación se presenta el *pseudo-código* de una implementación del algoritmo de Dijkstra que utiliza una cola de prioridad y la operación `decreaseKey`:

```
1  Dijkstra(Grafo, origen):
2  PriorityQueue Q
3  for vertice v in Grafo do
4  |  if v == origen then
5  |  |  dist[v] = 0 // distancia de origen a origen es 0
6  |  end
7  |  else
8  |  |  dist[v] = INFINITY // la distancia de origen a v es desconocida
9  |  end
10 |  prev[v] = null // predecesor de v
11 |  Q.add(v, dist[v]) // Agregar v a cola con su distancia a origen como prioridad
12 end
13 while Q is not empty do
14 |  m = Q.ExtractMin() // sacar vértice con mayor prioridad
15 |  // cada vecino que sigue en Q
16 |  for neighbor v of m do
17 |  |  nuevaDist = dist[m] + peso(m, v) // distancia de origen a v pasando por arista (m,v)
18 |  |  if nuevaDist < dist[v] then
19 |  |  |  dist[v] = nuevaDist // actualizar distancia a v
20 |  |  |  prev[v] = m // se pone a m como predecesor de v
21 |  |  |  Q.decreaseKey(v, nuevaDist) // aumenta prioridad de v
22 |  |  end
23 |  end
24 end
25 return dist, prev // retornar arreglo de distancias y caminos
```

2.2.1 Decrease Key

Esta operación disminuye el valor de una llave, así aumentando su prioridad. Su implementación difiere según el tipo de cola.

Heap Clásico Se cambia el valor de la llave indicada. Si se ha roto la propiedad de heap (padre debe tener menor valor que sus hijos), se hace swap de los elementos, subiendo por la estructura hasta que no se rompa el invariante o se llegue a la raíz. En caso de no romperse la condición, se retorna.

Fibonacci Heap Para llevar a cabo la operación Decrease Key, se añade una función nueva a la estructura: `cut(x)`. Este método corta el enlace entre x y su padre, separando el árbol. Se agrega la restricción de no poder cortar más de dos hijos de un nodo (exceptuando la raíz), por lo que cuando cortamos un enlace entre un nodo y su padre, marcamos este último. Al cortar un hijo de un nodo ya marcado, se corta el enlace, se desmarca el padre y se procede a cortarlo

recursivamente, hasta llegar a un nodo no marcado o a la raíz del árbol.

`decreaseKey(k, v)` disminuye el valor de la llave k a v y se actualiza el puntero al elemento mínimo si es necesario. Si es que no se rompe el orden del heap, se retorna. En caso contrario, se corta el elemento del árbol, utilizando la función `cut(v)`.

3 Experimentos y Datos

Se pide comparar el tiempo de ejecución del algoritmo de Dijkstra. Para esto tendrá que correr las tres implementaciones en grafos con distinta cantidad de aristas, explicados más adelante. Recuerde repetir los experimentos para cada configuración de grafo (n, e) de manera de obtener promedios confiables. Determine, de acuerdo a las repeticiones que haga de sus experimentos, el error asociado a los promedios para las mediciones que realice y utilícelo en sus gráficos.

Datos

Construya grafos conexos de $n = 100.000$ nodos y $e = \{10n, 100n, 1000n\}$ aristas, de la siguiente forma:

```
1 for i ← 1...n − 1 do
2 |  crear arista que lo conecte con siguiente nodo //asegurar conectividad
3 end
4 agregar e-n+1 aristas al azar
5 for arista in E do
6 |  weight(arista) = random(0,1) //se asignan pesos, 0 no incluido
7 end
```

Análisis

Discuta y muestre los resultados obtenidos. ¿Cómo se comparan los rendimientos? ¿Cómo varía el tiempo de ejecución según la cantidad de aristas? ¿Cómo se comparan sus resultados experimentales con la complejidad teórica de las implementaciones? Comente la razón de estas respuestas, basándose en lo aprendido en el curso y su conocimiento. Incluya otras observaciones o comentarios que encuentre pertinente.

Contraste sus conclusiones con las hipótesis planteadas.

4 Entrega de la tarea

- La tarea puede realizarse en grupos de a lo más 3 personas.
- El código debe ser de **implementación propia**.
- No se puede utilizar librerías para la implementación de las soluciones.

- Para la implementación solo puede utilizar C++, C o Java. Para el informe se recomienda utilizar LaTeX.
- Siga buenas prácticas (*good coding practices*) en sus implementaciones.
- Escriba un informe claro y conciso (no más de 10 páginas). Las ponderaciones del informe y la implementación en su nota final son las mismas.
- Tenga en cuenta las sugerencias realizadas en las primeras clases sobre la forma de realizar y presentar experimentos.
- La entrega será a través de U-Cursos y deberá incluir el informe junto con el código fuente de la implementación, y todas las indicaciones necesarias para su ejecución en un archivo `README.md`.