Grafos etiquetados

Algoritmos en grafos ponderados

- Existen muchas situaciones donde además del modelo de grafo, se requiere además asociar un peso/longitud a las artistas del grafo.
- Estos grafos se denominan grafos ponderados, grafos etiquetados.
- Importantes problemas prácticos se abordan por medio de grafos ponderados. Por ejemplo:
  - Encontrar los caminos más cortos en un grafo
  - Encontrar el árbol de cubrimiento mínimo

Grafos **no dirigidos** con pesos

`EdgeWeightedGraph(V)`	// Constructor
`int V()`	// Número de vértices
`int E()`	// Número de aristas
`void addEdge(Edge e)`	// Agregar una arista
`Iterable<Edge> adj(int v)`	// Aristas adyacentes a v
`int degree(int v)`	// Grado de un vértice

Implementación

`Edge(int v, int w, double w)`	// Constructor
`int either()`	// Uno de los vértices
`int other(int v)`	// El otro vértice
`double weight()`	// Peso de la arista

Implementación

Grafos **dirigidos** con pesos

`EdgeWeightedDigraph(V)`	// Constructor
`int V()`	// Número de vértices
`int E()`	// Número de aristas
`void addEdge(Edge e)`	// Agregar una arista
`Iterable<DirectedEdge> adj(int v)`	// Aristas adyacentes a v
`int outdegree(int v)`	// Grado saliente de un vértice
`int indegree(int v)`	// Grado entrante a un vértice

Implementación

`DirectedEdge(int v, int w, double w)`	// Constructor
`int from()`	// Vértice origen de la arista
`int to()`	// Vértice destino de la arista
`double weight()`	// Peso de la arista

Implementación

Caminos más cortos

- En un grafo dirigido con pesos positivos, interesa encontrar el camino más corto desde un origen *s* a un nodo destino t.
- Una de las soluciones más conocidas es el algoritmo de Dijkstra.
- Este algoritmo encuentra las rutas más cortas de *s* a los demás nodos.

Algoritmo de Dijkstra (1)

- Se mantienen tres estructuras auxiliares:
  - `distTo[i]`: Longitud del camino más corto de *s* a *i*.
  - `edgeTo[i]` : Última arista en el camino de *s* a *i*. Inicializado en `null`.
  - pq : Cola de prioridad mínima con los vértices indexados en función de su distancia al origen. (Ver `IndexMinPQ`).

```
public class DijkstraSP {
    private double[] distTo;
    private DirectedEdge[] edgeTo;
    private IndexMinPQ<Double> pq;

    …
}
```

Sobre cola de prioridad indexada IndexMinPQ

- MinPQ solo contiene llaves comparables.
- En dijkstra necesitamos los nodos (n=0,...V-1) y sus respectivas distancias al origen, priorizados en función de la distancia al origen.
- Por ejemplo:

nodo	5	1	4	2	6
distancia	1.1	2.3	3.2	3.8	4.5

IndexMinPQ

- Similar a MinPQ, pero se mantienen 3 arreglos:

```
private int[] pq;    // indices de los nodos
private int[] qp;    // Posición en la cola del indice i
private Key[] keys;  // Las llaves (distancias en Dijkstra)
```

- Las operaciones se implementan considerando tanto los índices, como las llaves. La prioridad la determinan las llaves.

```
insert(int i, Key key)           // Insertar un par indice,llave
int minIndex()                   // El índice del menor par
Key minKey()                     // La menor llave
int delMin()                     // Borrar el menor par
Key keyOf(int i)                 // Llave correspondiente a un índice
void changeKey(int i, Key key)   // Cambiar una llave
void decreaseKey(int i, Key key) // Decrementar el valor de una llave
boolean contains(int i)          // Está el índice en la cola?
```

Algoritmo de Dijkstra (2)

- Inicialización
  - `distTo[i]` : Longitud de los caminos s→i inicialmente infinito.
  - `edgeTo[i]` : Inicialmente no se han identificado aristas en los caminos más cortos: `null`.
  - pq : El vértice origen *s*.

```
distTo = new double[G.V()];
edgeTo = new DirectedEdge[G.V()];

for (int v = 0; v < G.V(); v++)
  distTo[v] = Double.POSITIVE_INFINITY;
distTo[s] = 0.0;

pq = new IndexMinPQ<Double>(G.V());
pq.insert(s, distTo[s]);
```

Algoritmo de Dijkstra (3)

- Proceso de “relajación” de aristas:
  - Al considerar la arista v→w , si la longitud del camino a w pasando por v es más corta:
    - actualizar la `distTo[w]` con la nueva distancia
    - actualizar la última arista hasta w con la arista v→w.
    - actualizar la cola de prioridad acorde a la nueva distancia hasta w.

```
private void relax(DirectedEdge e) {
  int v = e.from(), w = e.to();
  if (distTo[w] > distTo[v] + e.weight()) {
    distTo[w] = distTo[v] + e.weight();
    edgeTo[w] = e;
    if (pq.contains(w)) pq.decreaseKey(w, distTo[w]);
    else                pq.insert(w, distTo[w]);
  }
}
```

Algoritmo de Dijkstra (4)

- El ciclo central del algoritmo procede mientras la cola de prioridad no este vacía.
- Se obtiene el menor vértice de la cola y se relajan todas las aristas salientes de este vértice.

```
while (!pq.isEmpty()) {
  int v = pq.delMin();
  for (DirectedEdge e : G.adj(v))
    relax(e);
}
```

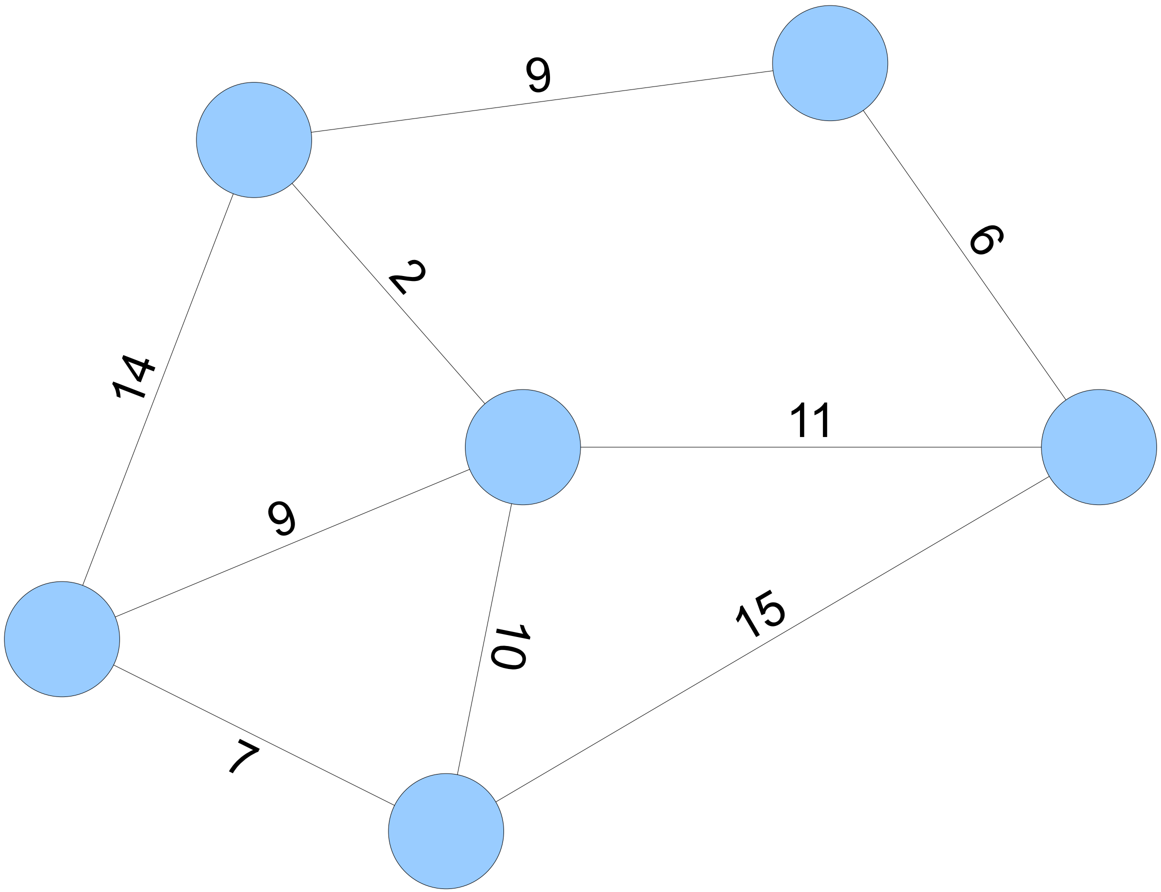
Implementación del algoritmo

Ejercicio

- Indicar el camino más corto del origen a cualquier nodo w:

```
void shortestPath(int w)
```

Dijkstra: Caso ejemplo



Eficiencia de Dijkstra

- La función relax(Edge e) realiza un número constante de operaciones aritméticas e invoca dos operaciones de la cola de prioridad:
  - `pq.contains(w)` : Que es de tiempo constante ~C y
  - `pq.decreaseKey(w,distTo[w])` o `pq.insert(w,distTo[w])`, que son ambas ~lg(V).
- El tiempo total por invocación de `relax` es:

~lg(V)

Eficiencia de Dijkstra (2)

- El ciclo `while` principal realiza dos operaciones:
- `pq.delMin()` : Borra el menor elemento de la cola de prioridad, como máximo el número de vértices del grafo: ~V lg(V).
- `relax(e)` para todas las aristas adyacentes, que totalizando sobre todas las iteraciones corresponden al número de aristas del grafo E, por lo que en total se tiene: ~E lg(V).
- El tiempo total es:

$$\sim(V+E)\ \lg(V)$$

Consideraciones

- Si el grafo es denso, $E \sim kV^2$. En este caso el tiempo total tiene la forma:

$$\sim cV^2 \lg(V)$$

- En el caso de grafos no densos, que es bastante común en la práctica, se tiene $E \sim kV$ y por lo tanto el tiempo total es:

$$\sim cV \lg(V)$$

Ejercicio

- Se puede modificar Dijkstra para encontrar la “ruta más larga” ?
- Bajo que consideraciones sería valida la solución propuesta? Qué grafos pueden llevar a problemas?

Otro ejercicio

- Obtener el camino más corto entre todos los pares de nodos (v,w)

Solución del texto guía

El problema del agente viajero Travelling Salesman Problem (TSP)

- Un agente viajero debe visitar N ciudades, cada ciudad una sola vez.
- El agente parte de una ciudad origen s y debe terminar en la misma ciudad: El recorrido es un ciclo.
- Las aristas representan las distancias entre ciudades.
- Se desea encontrar el recorrido de **menor distancia total**.

Características de TSP

- No se conoce ningún algoritmo exacto de tiempo polinómico.
- Tampoco se sabe si este algoritmo existe.
- Soluciones exactas conocidas requieren evaluar todos los posibles recorridos (todas las combinaciones de nodos). Llevan a tiempo exponencial.
- Es un problema de la clase NP-Complete: Problemas a los que no se les conoce solución de tiempo polinómico, pero que su solución es fácil de verificar.

Soluciones no exactas

Para problemas de la familia NP-Complete, usualmente se adoptan dos tipos de solución

1) Solución aproximada: De tiempo polinómico, pero con un margen de error conocido.

2) Heurística: Soluciones “intuitivas” pero que no tienen ninguna garantía de optimalidad.