Teoría de Algoritmos I

Trabajo Práctico n.º 1

El trabajo práctico consiste en las dos partes que se listan a continuación.

Lineamientos básicos:

- el trabajo se realizará en grupos de tres o cuatro personas.
- la fecha de entrega es el **lunes 10 de octubre de 2016**. Se debe entregar en el horario de clase en papel (informe + `código en monoespacio`), más una entrega en digital de código (.zip) e informe (.pdf) a la lista de docentes: XXX.
- el lenguaje de implementación es libre, pero se debe comunicar por correo en caso de *no* ser uno de: C, C++, Python, Java, JavaScript, Ruby, Go, Rust, Swift, Scala, Haskell, OCaml, Clojure, D, Lua, Elixir.

Contenidos

Estadístico de orden *k*

El estadístico de orden *k* de un conjunto de *n* elementos es el el *k*-ésimo elemento más pequeño. Son casos particulares de estos el máximo (*k* = *n-1*), el mínimo (*k* = *0*) y la mediana de un conjunto (*k* = *n/2*). Se propone analizar varios algoritmos que computan este valor.

1. **Fuerza bruta**: se implementa un verificador que dado un conjunto y un candidato devuelve un booleano indicando si el valor indicado es el *k* elemento más pequeño. El algoritmo de fuerza bruta itera todos los elementos del conjunto y verifica de a uno si es la solución. Una vez el verificador devuelve *true*, devuelve ese elemento.
2. **Ordenar y seleccionar**: ordena el conjunto mediante un algoritmo veloz de comparación y luego seleccionar el *k* elemento del arreglo ordenado.
3. **k-selecciones**: el algoritmo de ordenamiento por selección busca el menor elemento de una secuencia y lo intercambia con el primero. Se propone realizar *k* selecciones para encontrar el *k* elemento más pequeño.

4. **k-heapsort**: así como heapsort es una mejora del algoritmo de selección usando un heap, este algoritmo mejora el de k-selecciones haciendo *k* extracciones a un arreglo con la propiedad de heap.

5. **HeapSelect**: se propone usar un heap para almacenar los *k* elementos más chicos, intercambiándolos cuando sea necesario.

6. **QuickSelect** (Cormen cap. 10): se usa una estrategia de división y conquista similar a la de quicksort pero descartando las divisiones que sabemos que no incluyen al *k* buscado.

Se pide:

- Implementar todos los algoritmos propuestos.
- Calcular analíticamente la complejidad de cada uno (excepto la de QuickSelect).
- Mostrar conjuntos de 16 elementos con los mejores y peores casos de cada algoritmo.
- Graficar los tiempos de ejecución para entradas aleatorias de tamaño representativo.
- Analizar qué algoritmo es el óptimo para cada *k* según el tamaño de la entrada (*n*).

Recorridos en grafos

Se quiere resolver el problema de encontrar el camino más barato entre un par de nodos de un grafo dirigido, pesado y sin aristas negativas. Se proponen las siguientes alternativas:

1. **BFS**: dado un vértice de origen implementa el recorrido en anchura ignorando los pesos. Esta variación finaliza el algoritmo cuando se visita el destino.

2. **Dijkstra**: dado un vértice de origen y de destino implementa el algoritmo de Dijkstra de camino mínimo.

3. **Búsqueda con heurística**: similar a BFS, pero encola los vértices en una cola de prioridad según una heurística de acercamiento al destino.

   Esto es, ignora los pesos de las aristas pero consulta una función `heuristica(v)` que da una aproximación de la distancia entre un vértice del camino, y el destino final.

4. **A***: considera tanto la heurística como el peso de las aristas para encontrar el camino mínimo.

Se pide:

- Implementar el grafo respetando la API sugerida por el curso.
- Implementar los cuatro algoritmos de búsqueda propuestos como clientes de la API.
- Calcular analíticamente la complejidad de cada algoritmo, considerando que la heurística funciona en tiempo constante.
- Mostrar grafos de ejemplo en donde se obtengan los mejores y peores casos de cada algoritmo.
- Analizar bajo qué situaciones cada algoritmo calcula la solución óptima y es más eficiente. Siendo *h* la heurística y *R* el costo real desde *v:*
  - Grafos con y sin peso.
  - Heurísticas que funcionen sin error, es decir, que siempre estimen `h(v) = R(u)`.
  - Heurísticas consistentemente optimistas que siempre estimen `h(v) << R(v)`.
  - Heurísticas consistentemente pesimistas que siempre estimen `h(v) >> R(v)`.

Notas de implementación

En general, recomendamos seguir un patrón de diseño parecido al de Robert Sedgewick en *Algorithms* (2011, 4.ª ed.).

En primer lugar, TAD Grafo inmutable de funcionalidad mínima, y en segundo lugar un TAD separado para la búsqueda de caminos, con cuatro implementaciones diferentes.

El TAD Grafo mantendrá la mínima representación necesaria del grafo. Como es dirigido y pesado, se recomienda usar una pequeña clase *Arista* para guardar los pesos.

En cuanto a los algoritmos, se recomienda:

- mantener una interfaz común en los cuatro, que proporcione los siguientes métodos:
  - constructor que acepte: grafo, origen, destino (más heurística, cuando corresponda)
  - `visitado(v)`: si se visitó un determinado vértice durante el recorrido
  - `distancia(v)`: la distancia hasta un vértice visitado (si no fue visitado, devolver ∞)
  - `camino(v)`: la lista de aristas desde el origen hasta un vértice (puede devolver *null* si el vértice no fue visitado, y una lista vacía en caso de pasarse el vértice de origen)

- de querer usar herencia, implementar en una clase base las funciones *visitado* y *camino*, implementadas ambas usando los métodos abstractos:
  - `distancia(v)` (explicado arriba)
  - `_edge_to(v)`: la arista que conduce a un vértice *v* en el camino de origen a destino
- realizar todo el trabajo de cada algoritmo en su propio constructor (así, una vez inicializado, es inmutable y se convierte en un objeto de solo-consulta).

Como ejemplo en Python, el TAD Grafo podría plantearse así:

```
class Digraph:
  """Grafo no dirigido con un número fijo de vértices.

  Los vértices son siempre números enteros no negativos. El primer
  es 0.

  El grafo se crea vacío, se añaden las aristas con add_edge(). Una
  creadas, las aristas no se pueden eliminar, pero siempre se puede
  nuevas aristas.
  """
  def __init__(g, V):
    """Construye un grafo sin aristas de V vértices.
    """

  def V(g):
    """Número de vértices en el grafo.
    """

  def E(g):
    """Número de aristas en el grafo.
    """

  def adj_e(g, v):
    """Itera sobre los aristas incidentes _desde_ v.
    """

  def adj(g, v):
    """Itera sobre los vértices adyacentes a 'v'.
    """

  def add_edge(g, u, v, weight=0):
    """Añade una arista al grafo.
    """
```

```
  def __iter__(g):
    """Itera de 0 a V."""
    return iter(range(g.V()))

  def iter_edges(g):
    """Itera sobre todas las aristas del grafo.

    Las aristas devueltas tienen los siguientes atributos de solo l

        • e.src
        • e.dst
        • e.weight
    """

class Arista:
  """Arista de un grafo.
  """
  def __init__(self, src, dst, weight):
    ...
```

Como ejemplo en Java, los algoritmos podrían representarse así:

```
public abstract class Caminos {

    private int src;

    protected Caminos(Digraph g, int origin) {
        src = origin;
    }

    public abstract double distancia(int v);

    protected abstract Arista edge_to(int v);

    public boolean visitado(int v) {
        return distancia(v) < Double.POSITIVE_INFINITY;
    }

    public List<Arista> camino(int v) { /* ... */ }
}

public class BFS extends Caminos {

    private double dist[];  // Inicializar a +∞.
```

```
    private Arista edge[];

    public BFS(Digraph g, int origin, int destino) {
        super(g, origin);

        // código del algoritmo, rellena "dist" y "edge".
    }

    public double distancia(int v) { return dist[v]; }
    protected Arista edge_to(int v) { return edge[v]; }
}
```