Universidad de Buenos Aires

Facultad de Ingeniería

75.29 Teoría de Algoritmos

Trabajo Práctico 1

Integrantes

- Arjovsky, Tomás
- Gavrilov, Seva
- Pereira, Fernando
- Pérez Dittler, Ezequiel

Segundo cuatrimestre de 2016

Índice

1. Código

Aclaración: en este documento solo aparece el código que refleja las consignas del trabajo práctico. En nuestro repositorio de código se pueden encontrar tests, ejemplos, benchmarks de algoritmos y más: https://github.com/ezeperez26/tda-fiuba

1.1. K-estadístico

1.1.1. Brute force

```python
def brute_force(l, k):
    """
    Estadístico de orden k por fuerza bruta.
    l es el conjunto de elementos,
    k es el orden del elemento más chico.
    """

    # función interna que verifica si un elemento es el k más chico
    def _is_k_lowest(elem):
        lower_elements_count, equal_elements_count = 0, 0
```

```
        for item in l:
            if item < elem:
                lower_elements_count += 1
            elif item == elem:
                equal_elements_count += 1

        # Si la cantidad de elementos más pequeños es q y k está entre
        # q y q + la cantidad de elementos iguales, elem es el k más pequeño
        return lower_elements_count <= k < (lower_elements_count + equal_elements_count)

    for item in l:
        if _is_k_lowest(item):
            return item
```

1.1.2. Order and Select

```
def order_and_select(l, k):
    """
    Se ordena el conjunto dado y se devuelve el k-esimo valor
    """

    list.sort(l)
    return l[k]
```

1.1.3. K-selecciones

```
def k_selections(l, k):
    """
    Estadistico de orden k por k-selecciones. l es el conjunto de elementos, k
    es el orden del elemento mas chico que se quiere encontrar.
    """

    # selection sort parcial (solo los primeros K)
    for index in range(0, k + 1):
        min_value_index = index
        min_value = l[index]
        for inner_index in range(index + 1, len(l)):
            if l[inner_index] < min_value:
                min_value_index = inner_index
                min_value = l[inner_index]
        l[min_value_index], l[index] = l[index], l[min_value_index]
    return l[k]
```

1.1.4. K-heapsort

```python
import heapq

def k_heapsort(elements, k):
    """
    Se hace un heapify y luego se extraen k elementos.
    El último extraido es el k-min.
    """
    heap = elements.copy()
    heapq.heapify(heap)  # O(n) (https://docs.python.org/2/library/heapq.html#heapq.heapify)

    # Sacar k elementos y devolver el último. O(klog(n)) [O(nlog(n) en el peor caso]
    # Como k puede ser 0, el elemento que busco es k + 1
    item = None
    for _ in range(k + 1):
        item = heapq.heappop(heap)
    return item
```

1.1.5. HeapSelect

```python
import heapq


def heap_select(elements, k):
    """
    Se almancenan los k elementos más chicos en un heap de máximo y
    se devuelve el k-esimo valor
    """
    minimos = [(-i, i) for i in elements[:k+1]]
    heapq.heapify(minimos)

    for elem in elements[k+1:]:
        # Se inserta el elemento si y solo si es menor al máximo del heap.
        # En el caso de insertarlo, se saca el máximo.
        if elem < minimos[0][1]:
            heapq.heappush(minimos, (-elem,elem))
            heapq.heappop(minimos)

    return minimos[0][1]
```

1.1.6. QuickSelect

```
from math import floor
import random


def quick_select_algorithm(l, k, first_index, last_index):
    """
    Mismo 'paradigma' que quicksort. Se elige pivote, se ordena el pivote
    y se chequea numero de indice. Dependiendo del mismo (y de k) se elige
    que lado procesar
    """

    # Funcion auxiliar que me ordena el pivote en la lista
    def order_pivot(l, first_index, last_index, pivot_index):
        pivot_value = l[pivot_index]
        l[pivot_index], l[last_index] = l[last_index], l[pivot_index]
        final_pivot_index = first_index

        for index in range(first_index, last_index):
            if l[index] < pivot_value:
                l[index], l[final_pivot_index] = l[final_pivot_index], l[index]
                final_pivot_index += 1

        l[last_index], l[final_pivot_index] = l[final_pivot_index], l[last_index]

        return final_pivot_index


    #for index in range(first_index, last_index):
    while(True):
        if first_index == last_index:
            return l[first_index]

        # Elijo el pivote arbitrariamente y opero
        pivot_index = floor((first_index + last_index) / 2)
        pivot_index = order_pivot(l, first_index, last_index, pivot_index)

        if k == pivot_index:
            return l[k]
        elif k < pivot_index:
            last_index = pivot_index - 1
        else:
            first_index = pivot_index + 1
```

1.2. Camino mínimo en Grafos

1.2.1. Implementación del grafo

```
class Graph(object):
    """Grafo no dirigido con un número fijo de vértices.

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

    El grafo se crea vacío, se añaden las aristas con add_edge(). Una vez
    creadas, las aristas no se pueden eliminar, pero siempre se puede añadir
    nuevas aristas.
    """

    def __init__(g, V):
        """ Construye un grafo sin aristas de V vértices. """
        g._a = [[] for _ in range(V)]  # _a es la matriz de adyacencias
        g._i = [[] for _ in range(V)]  # _i es la matriz de incidencias

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

    def E(g):
        """ Número de aristas en el grafo. """
        return sum(len(x) for x in g._i)

    def adj_e(g, v):
        """ Itera sobre los aristas incidentes desde v. """
        return iter(g._i[v])

    def adj(g, v):
        """ Itera sobre los vértices adyacentes a v. """
        return iter(g._a[v])

    def add_edge(g, u, v, weight=0):
        """ Añade una arista al grafo. Devuelve la arista agregada """
        a = Edge(u, v, weight)
        g._i[u].append(a)
        g._a[u].append(v)
        return a

    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 lectura:
            - e.src
            - e.dst
            - e.weight
        """
        return iter(edge for edges in g._a for edge in edges)

    def has_node(g, v):
        return v < len(g._a)

    @classmethod
    def from_dict(cls, d):
        """ Toma un diccionario, compuesto de la siguiente manera:
            {
                nodo: [nodo_adjacente, nodo_adjacente2],
                nodo2: [...]
            }
            Por ejemplo:
            {
                1: [2, 3, 4, 5]
                2: [3],
                4: [5]
            }

            El listado de nodos adyacentes tambien puede ser una tupla, donde
            el primer elemento es el nodo y el segundo es el peso de la arista.
        """
        max_ady = max(v[0] if isinstance(v, tuple) else v
                      for values in d.values() for v in values)
        max_node = max(max(d.keys()), max_ady)
        g = cls(max_node + 1)
        for src, v in d.items():
            for dst in v:
                weight = 1
                if isinstance(dst, tuple):
                    weight = dst[1]
                    dst = dst[0]
                if g.has_node(dst):
                    g.add_edge(src, dst, weight)
        return g


class Edge(object):
    """ Arista de un grafo. """

    def __init__(self, src, dst, weight):
        self.src = src
```

```
        self.dst = dst
        self.weight = weight


def create_grid_graph(x, y):
    """
    Devuelve un grafo grilla de x por y, y una heurística que devuelve la
    distancia entre 2 nodos.
    """
    max_node = x * y
    points = {}
    g = Graph(max_node)
    for i in range(x):
        for j in range(y):
            node = i + j * x
            points[node] = (i, j)
            if i - 1 >= 0:
                g.add_edge(node, (i - 1) + j * x)
            if i + 1 < x:
                g.add_edge(node, (i + 1) + j * x)
            if j - 1 >= 0:
                g.add_edge(node, i + (j - 1) * x)
            if j + 1 < y:
                g.add_edge(node, i + (j + 1) * x)

    def heuristic(v, u):
        return abs(points[v][0] - points[u][0])  + abs(points[v][1] - points[u][1])

    return g, heuristic
```

1.2.2. Implementación de camino

```


class CommonPath(object):
    """
    Abstracción de búsqueda de caminos para todos los algoritmos.

    Los algoritmos deberán ser subclase de CommonPath y definir el método
    search, que se va a llamar en el constructor.
    La idea (inicial) es que se llene el diccionario parents con vertices como
    claves y valores. Ese diccionario luego se va a usar para reproducir los
    caminos y calcular las distancias.
    Si se le especifica search=False, no va a realizar la búsqueda en el
    constructor y se tendrá que correr manualmente.
    """

    def __init__(self, g, u, v, heuristic=None, search=True):
```

```
        """
        g es el grafo a trabajar, u es el nodo origen, v es el nodo destino
        """
        self.g, self.u, self.v = g, u, v
        self.heuristic = heuristic
        self.parents = {}
        self._visited = set()
        if search:
            self.search()

    def search(self):
        """ Este método deberá ser redefinido en subclases. """
        pass

    def visited(self, v):
        return v in self._visited

    def distance(self, v):
        return len(self.path_to(v))

    def path_to(self, v):
        """
        Devuelve la lista con el camino hasta el nodo pasado a partir del
        diccionario parents. Si el nodo no fue visitado, devuelve None.
        """
        if self.visited(v):
            path = [v]
            while path[-1] != self.u:
                path.append(self.parents[path[-1]])
            path.reverse()
            return path
```

1.2.3. BFS

```
from path import CommonPath


class BFS(CommonPath):
    """
    Realiza una búsqueda BFS, cortando al encontrar el nodo destino.
    """

    def search(self):
        q = [self.u]

        while q:
            u = q.pop()
```

```
            self._visited.add(u)
            for v in self.g.adj(u):
                if v not in self.parents:
                    self.parents[v] = u
                    q.append(v)
                if v == self.v:
                    self._visited.add(self.v)
                    return
```

1.2.4. Dijkstra

```
from path import CommonPath
import math
import heapq


class _PriorityQueueNode():
        def __init__(self, distance, node):
            self.distance = distance
            self.node = node

        def __lt__(self, other):
            return self.distance <= other.distance


class Dijkstra(CommonPath):
    """
    Implementacion del algoritmo de Dijkstra para encontrar camino minimo
    usando priority queue
    """

    def search(self):

        distance = {}
        priority_queue = []

        # Inicializo todos los vertices con distancia infinita
        for v in self.g:
            distance[v] = math.inf

        # Inicializo nodo origen y lo agrego a la PQ
        distance[self.u] = 0
        heapq.heappush(priority_queue, _PriorityQueueNode(distance[self.u], self.u))

        # Voy procesando tomando los vertices con menor distancia desde origen
        while priority_queue:
            u = heapq.heappop(priority_queue)
```

```
            # Corroboro que el nodo no haya sido visitado
            # (Puede que se haya insertado dos veces en la PQ por haber
            # encontrado otro camino minimo que el primero encontrado)
            if not self.visited(u.node):

                # Condicion de corte -> Mi nodo actual es el destino
                if u.node == self.v:
                    self._visited.add(u.node)
                    return

                # Verifico todas las aristas salientes para computar distancias
                for edge in self.g.adj_e(u.node):
                    current_distance = u.distance + edge.weight
                    if (current_distance < distance[edge.dst]):
                        distance[edge.dst] = current_distance
                        self.parents[edge.dst] = u.node

                        # Actualizo la distancia minima hacia este vertice en la PQ
                        to_insert_node = _PriorityQueueNode(current_distance, edge.dst)
                        heapq.heappush(priority_queue, to_insert_node)

                self._visited.add(u.node)
```

1.2.5. Heurísticas

```
import heapq
from path import CommonPath


class Heuristic(CommonPath):
    """
    Realiza una búsqueda con heurística.
    """

    def search(self):
        queue = []
        heapq.heappush(queue, (0, self.u))

        while queue:
            u = heapq.heappop(queue)[1]
            self._visited.add(u)
            nodes = [(self.heuristic(v, self.v), v) for v in self.g.adj(u)]
            for tuple in nodes:
                v = tuple[1]
                if v not in self.parents:
                    self.parents[v] = u
```

```
                    heapq.heappush(queue, tuple)
                if v == self.v:
                    self._visited.add(v)
                    return
```

1.2.6. A*

```
from path import CommonPath
import math


class MinPQ(object):
    def __init__(self):
        self.heap = []

    def pop(self):
        return heapq.heappop(self.heap)

    def push(self, priority, element):
        heapq.heappush(self.heap, (priority, element))

    def empty(self):
        return self.heap == []


class A_Star(CommonPath):
    """
    Realiza una búsqueda teniendo en cuenta tanto el peso de las aristas
    como la heurística.
    """

    # Función de evaluación. Será la prioridad para la frontera.
    def f(self, v):
        return self.distance[v] + self.heuristic(v, self.v)

    def search(self):
        # distancias a cada nodo.
        self.distance = {}
        frontera = MinPQ()

        # Inicializo todos los vertices con distancia infinita
        for v in self.g:
            self.distance[v] = math.inf

        # Inicializo nodo origen y lo agrego a la PQ
        self.distance[self.u] = 0
        frontera.push(0, self.u)
```

```
        # Voy procesando tomando los vertices con menor distancia desde origen
        while not frontera.empty():
            (f, u) = frontera.pop()

            # Condicion de corte -> El nodo a visitar es el destino.
            if u == self.v:
                self._visited.add(self.v)
                return

            if not self.visited(u):
                self._visited.add(u)
                # Verifico todas las aristas salientes para computar distancias
                for edge in self.g.adj_e(u):
                    v = edge.dst
                    if not self.visited(v):
                        current_distance = self.distance[u] + edge.weight
                        if (current_distance < self.distance[v]):
                            self.distance[v] = current_distance
                            self.parents[v] = u
                            frontera.push(self.f(v), v)
```