Trabajo Práctico 1

Teoría de Algoritmos I (75.29)

Universidad de Buenos Aires

Facultad de Ingeniería

Segundo Cuatrimestre - 2016

Alumnos:

Emanuel Condo	(94773)
Martín Cura	(95874)
Fernando Nitz	(94994)
Nicolás Gago	(91677)

Estadístico de orden *k*

Implementación y análisis de distintos algoritmos para obtener el *k*-ésimo elemento más pequeño de un conjunto. En todos se toman algunas precondiciones implícitas, como que *k* sea un entero entre 1 y *n*, siendo *n* el tamaño del conjunto.

Cálculos analíticos de complejidad

Fuerza bruta

Se presentan 2 versiones de este algoritmo, con uno más simple que asume que si el valor está repetido su *k* a verificar es el mínimo. Ambos poseen la misma complejidad.

Este algoritmo recorre el conjunto entero en orden hasta encontrar más de *k* números cuyo valor es mayor al del candidato, demostrando expresamente que su verdadero *k* es en realidad mayor. En caso de no superar este número pero tampoco quedar corto, verifica que el *k* era el correcto. Para que este algoritmo sirva el mismo propósito que los demás, y salvando alguna heurística para acortar esta iteración total, se debe probar cada valor del conjunto (*n* valores) hasta encontrar uno que devuelva True.

Analíticamente, la complejidad global será equivalente a:

```
for i = 1 to n:
    for j = 1 to n:
          if ( ... )
                return
```

con todo lo demás de menor orden, por lo que en el caso promedio el primer *loop* recorrerá la mitad del conjunto hasta encontrar el candidato correcto y el segundo *loop* tenderá a demostrar que cierto *cand* no es el buscado en (también) medio recorrido, por lo que tenemos que la complejidad será, tanto en el *average-case* como en el *worst-case*,

$$T(n) = O(n/2 * n/2) = O\left(n^2 / 4\right) = O\left(n^2\right)$$

El mejor caso para este algoritmo es encontrar el *cand* en el primer *outer loop*, y esto ocurrirá si este es el primero probado por la implementación; en el caso de *k_fuerza_bruta* esto ocurre si el primer elemento del conjunto es el *k* buscado. Un ejemplo sería trivial. Pero el mejor caso considerando solo el segundo loop para cualquier candidato dentro del conjunto se logra si este se encuentra ordenado como el estadístico buscado, esto es, de menor a mayor, ya que esto reduce el *inner loop* lo más posible para todos los casos donde se pruebe falso. Cualquier conjunto así ordenado sirve como ejemplo. Nótese que sin una heurística en el probado de valores de *cand*, si se permite un *k* cualquiera el mejor caso

sigue siendo $O(n^2)$. Para un *input* donde *k* será 1 y el primer elemento del conjunto es el buscado, el *running-time* respetará $O(n)$.

Ejemplo:

```
mejor_caso: conj = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16];k = 1
```

El peor caso ocurrirá si, inversamente, el conjunto se encuentra ordenado al revés, en este caso, de mayor a menor, ya que los valores con un menor estadístico son los últimos en ser chequeados, y si el candidato buscado es el último elemento del conjunto, lo cual en un conjunto ordenado de esta manera corresponde a *k = 1*. Nuevamente, la complejidad sigue siendo la misma.

Ejemplo:

```
peor_caso:  conj = [16,15,14,13,12,11,10,9,8,7,6,5,4,3,2,1];k = 1
```

Ordenar y seleccionar

La implementación aprovecha el `sort()` por defecto de Python, y después simplemente devuelve el *k*-ésimo elemento. Por lo tanto, su complejidad será la misma que la de este sort, el cual se basa en TimSort. Este posee un rendimiento de $O(n \log n)$ para el peor caso, $O(n)$ para el mejor, y $O(n \log n)$ para el *average-case*. Es por esto que este algoritmo presenta el mejor rendimiento de todos los expuestos.

Como este algoritmo toma provecho de subsecuencias ya ordenadas, el peor caso se puede buscar con un conjunto totalmente desordenado, donde el *randomness* es máximo (ir a discutir con Luisito si esto es posible). Por otro lado, el mejor caso ocurre si el conjunto ya se encontraba totalmente ordenado.

K-selecciones

Este algoritmo se comporta como un algoritmo de ordenamiento parcial que selecciona el menor elemento de un conjunto, es decir, realiza las iteraciones necesarias hasta encontrar el k-ésimo elemento buscado. Tomando un conjunto de elementos de tamaño *n,* con un *n* representativo, donde dependiendo del elemento k-ésimo a buscar, se obtendrán complejidades diferentes. En general, la complejidad del algoritmo es $O(kn)$.

Si *k* es lo suficientemente “chico”, en comparación a *n,* el algoritmo tendrá una complejidad lineal $O(n)$.

Si *k* es lo suficientemente “grande”, cercano a *n,* el algoritmo tendrá una complejidad cuadrática $O(n^2)$.

Analíticamente, la complejidad global será equivalente a (se están obviando las operaciones que poseen costo $O(1)$):

```
for i = 1 to k:
    for j = 1 to n:
          ...
```

$$T(n) = O(k) * O(n) * O(1) = O(kn)$$

El mejor caso para este algoritmo es encontrar el k-ésimo elemento mínimo del conjunto en la primera posición (con k = 1) en un conjunto ordenado, con lo cual la complejidad del algoritmo será $O(n)$.

```
mejor_caso: conj = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16];k = 1
```

Para el peor caso, el conjunto se encuentra ordenado de manera descendente y el k-ésimo elemento mínimo que se desea buscar es *k = n,* con lo cual la complejidad del algoritmo será $O(n^2)$.

```
peor_caso:  conj = [16,15,14,13,12,11,10,9,8,7,6,5,4,3,2,1];k = n
```

k-heapsort

Nuestro *k_heapsort* heapifica y retira un elemento del *heap* de mínimo sucesivas $(k-1)$ veces. Tras esto, un último *heapify* y el primer elemento será el buscado ya que se han sacado los $(k-1)$ que eran menores a él. La forma de "sacar" elementos es intercambiándolos con el último y modificando cuál es el final del *heap*, manteniendo el algoritmo como *in-place*. Analíticamente, su complejidad global es equivalente a:

```
for i = 1 to k:
    for j = n/2 to 1:
          moveDown()
```

El `moveDown()` a su vez tiene una complejidad de $O(log\ n)$, ya que recorre una única rama (entera en su peor caso) y la propiedad de *heap* le permite acceder a los hijos en $O(1)$. Por lo tanto, la complejidad del algoritmo entero se puede reducir a:

$$T(n) = O(k) * O(n/2) * O(log\ n) * O(1) = O(k\ n/2\ log\ n) = O(k\ n\ log\ n).$$

Está claro por esto que el peor caso se obtendrá por *k* grandes y un conjunto que requiera a su vez el peor caso para heapificar. Con *k* cercanos a *n*, los *loops* externos proveen una complejidad de $O(n^2)$, volviéndolo bastante ineficiente.

Un caso bueno de este algoritmo ocurre si hay un armado más rápido del *heap* cosa que ocurre cuando se encuentra ordenado el conjunto, de forma que el `moveDown()` inicial no tiene que hacer intercambios, y con *k* chica para reducir los siguientes. Ejemplo:

```
mejor_caso: conj = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16];k = 1
```

Por el contrario, se logra un mal caso si se maximiza esta cantidad de intercambios iniciales y, en mayor parte, si *k* se aproxia a *n* (esto es ya que el $n > log\ n$).

```
peor_caso:  conj = [16,15,14,13,12,11,10,9,8,7,6,5,4,3,2,1];k = 16
```

HeapSelect

En este algoritmo, se crea un *max-heap* de *k* elementos agregando nuevos si son menores al máximo (desplazando a este); el resultado será un *heap* cuyo máximo será el *k*-ésimo del conjunto original. Esto se logra con un recorrido de todo el conjunto, en los primeros *k* elementos con `siftUp()` y con el resto con `moveDown()`, ambas tienen *worst-case* en el orden de $O(log\ n)$. Reducido:

```
for i = 1 to k-1:
    siftUp()
for i = k to n:
    if (...)
          moveDown()
```

Lo cual se traduce a

$$T(n) = O(k) * O(log\ n) + O(n-k) * O(log\ n) = O(n) * O(log\ n) = O(n\ log\ n)$$

Para alcanzar un buen caso, nuevamente, armar el *heap* debe ser rápido, por lo que los primeros *k* elementos deben ser decrecientes, y después alcanza con que los siguientes ($n-k$) elementos sean mayores al que se encuentra en la raíz del *heap* para que nunca lo reemplace y esa sección toma $O(n-k)$. Si se busca menor gasto espacial también, *k* debería ser chico. Inversamente se pueden pensar en un peor caso.

```
mejor_caso: conj = [1,9,10,11,2,4,5,15,16,7,12,3,13,6,14,8];k = 1

peor_caso:  conj = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16];k = 16
```

QuickSelect

Este algoritmo está basado en el algoritmo quicksort, el cual tiene una buena performance promedio. Esto último depende del pivote elegido.

El mejor caso para este algoritmo es, primeramente, encontrar un buen pivote el cual permite descartar el subconjunto que no contiene el k-ésimo elemento buscado, por ende, el conjunto de búsqueda decrece y se realiza de manera lineal. El conjunto debe estar desordenado. Entonces la complejidad del algoritmo será $O(n)$.

Para el peor caso, el conjunto se encuentra ordenado por ende el pivote a elegir estará en un extremo, entonces en cada partición del conjunto sólo se descartá un subconjunto de un elemento lo cual no es eficiente. Al final, la complejidad del algoritmo será $O(n^2)$.

Comparación de tiempos de ejecución

En la Figura 1 graficamos los tiempos de ejecución medidos.

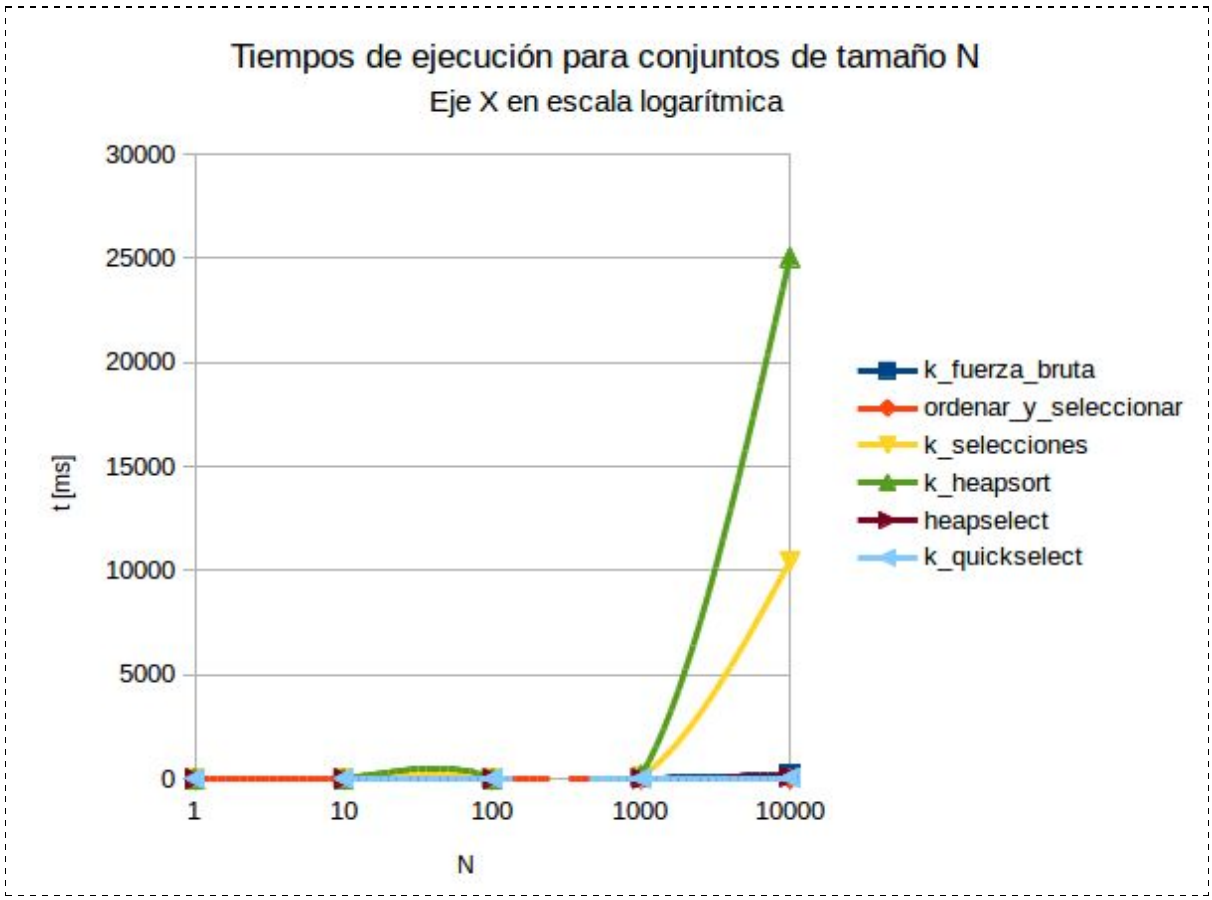


Figura 1: Gráfico comparativo de los tiempos de ejecución

Se puede ver que al crecer *n*, un par de algoritmos se alejan rápidamente del resto. Estos serán los menos eficientes para grandes *n*, y concuerdan con las complejidades determinadas anteriormente. Por otro lado, no dejan ver el rendimiento de los otros algoritmos, por lo que volvemos a graficar de manera logarítmica para el eje Y, así relativizando los grandes números y dejando ver las diferencias, en la Figura 2.

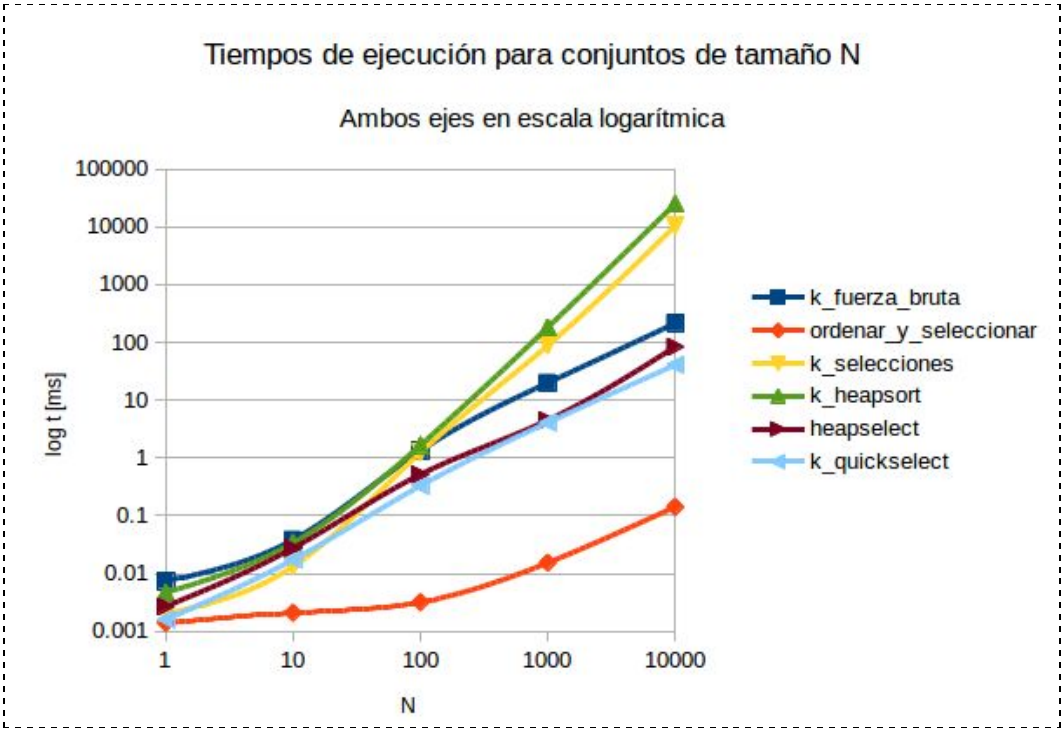


Figura 2: Gráfico comparativo de los tiempos de ejecución en escala logarítmica

Acá se pueden contrastar los resultados más claramente. Como puede verse, tanto k_heapsort y k_selecciones, con sus tiempos de corrida calculados previamente en el orden de $O(n^2 \log n)$ y $O(n^2)$ respectivamente para *k* grandes, rápidamente se alejan del resto, mientras que el ordenar_y_seleccionar se torna inmediatamente en el algoritmo más conveniente, ya que depende de un *sort* muy eficiente. Está claro que no se puede desestimar la complejidad si se esperan *n* muy grandes.

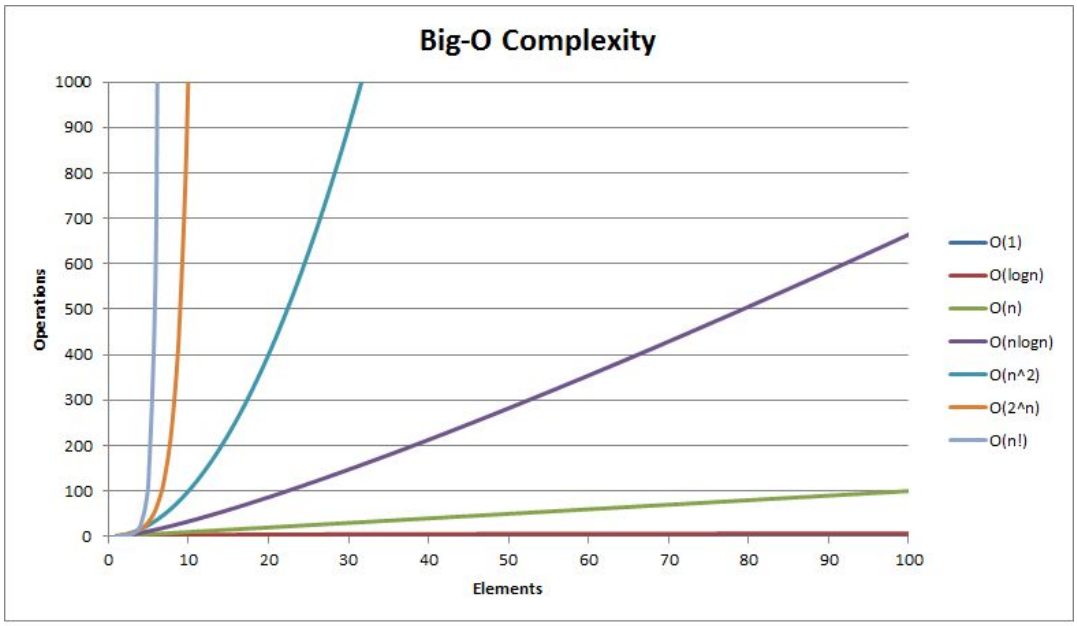


Figura 3: Gráfico comparativo de los órdenes de complejidad, robado de internet[1]

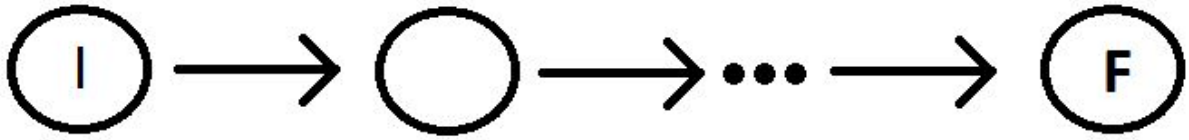
Como se puede ver en la Figura 3, para los crecientes *n* los tiempos de ejecución se alejarán rápidamente, y solo para un *n* relativamente chico puede considerar usarse otro que el mejor de los algoritmos (ordenar_y_seleccionar, gracias a su heredada $O(n \log n)$ en el caso promedio).

Debido a lo expuesto, se pueden sacar algunas conclusiones para valores de *k*. Para *k* chicos el caso promedio para aquellos con un *log* se aproxima a $O(n \log n)$, mientras que k_selecciones logra una complejidad de $O(n)$, volviéndose el claro ganador. Contrariamente, si *k* se aproxima más a *n*, su complejidad se vuelve $O(n^2)$, a la par de fuerza_bruta, pero todavía abajo del aún peor $O(n^2 \log n)$ de k_heapsort para esos casos.

[1] https://www.hackerearth.com/practice/notes/big-o-cheatsheet-series-data-structures-and-algorithms-with-thier-complexities-1/

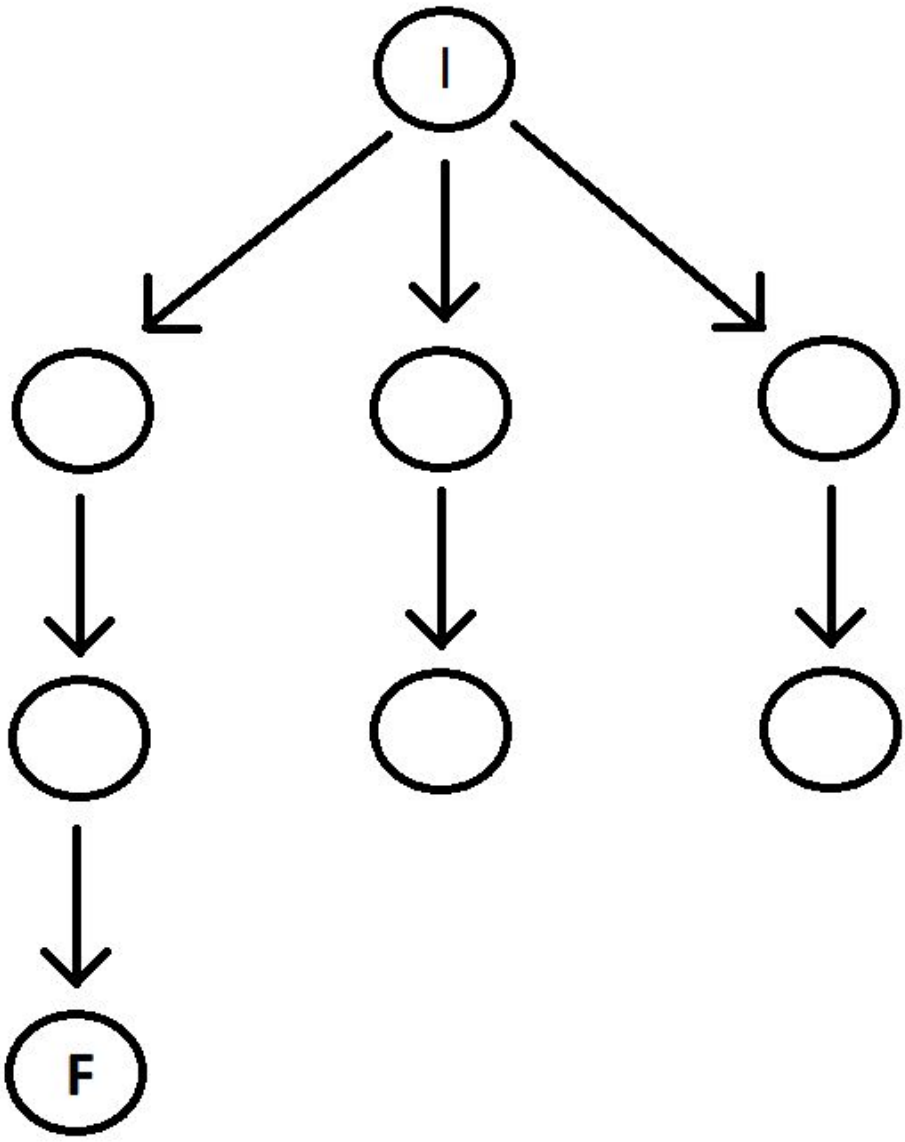
BFS y BFS con heurística:

Peor caso:
Como se puede notar, en el caso en el que un grafo funcione como una lista, y el nodo inicial y final se encuentren en los extremos de la lista, el algoritmo deberá recorrer todo el grafo para terminar.



(Caso 1)

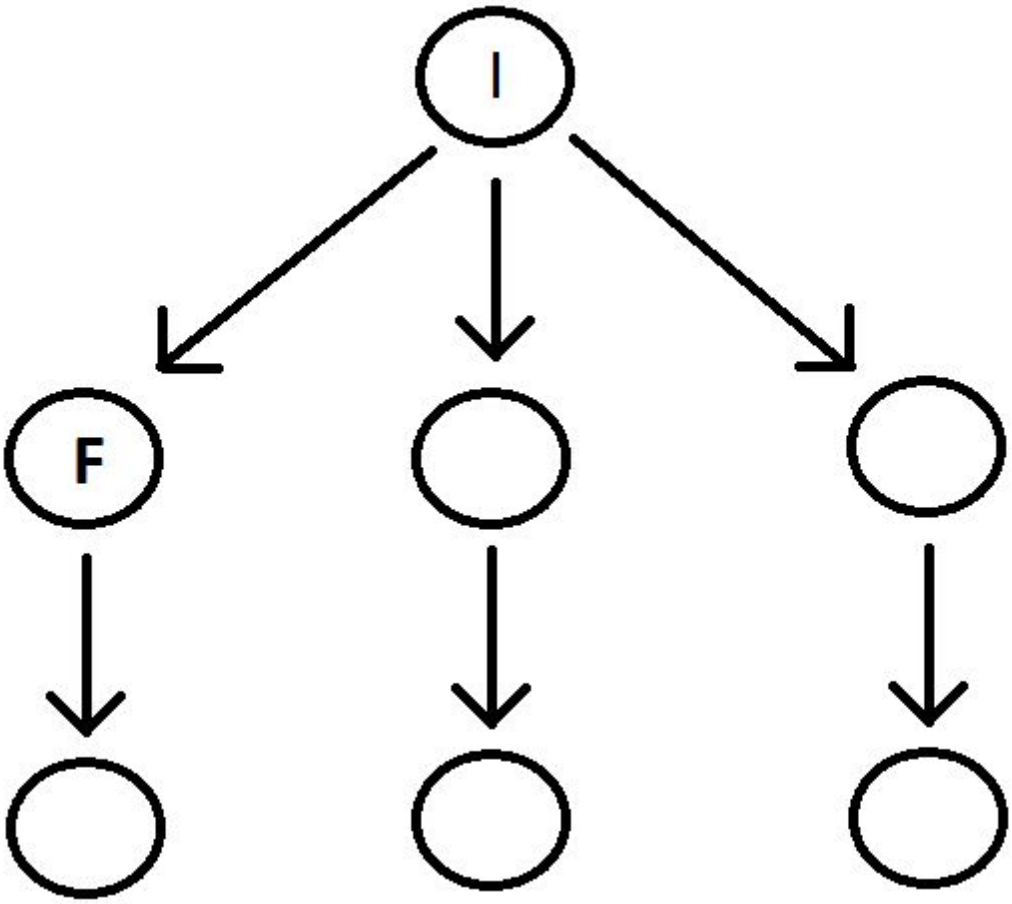
Otro peor caso, ocurre cuando el nivel del nodo final es el más grande (Denominado nivel a la cantidad de nodos que separan al nodo inicial del final).



(Caso 2)

Como se puede apreciar el BFS busca por nivel, por lo tanto, es innato al algoritmo ambos peores casos.

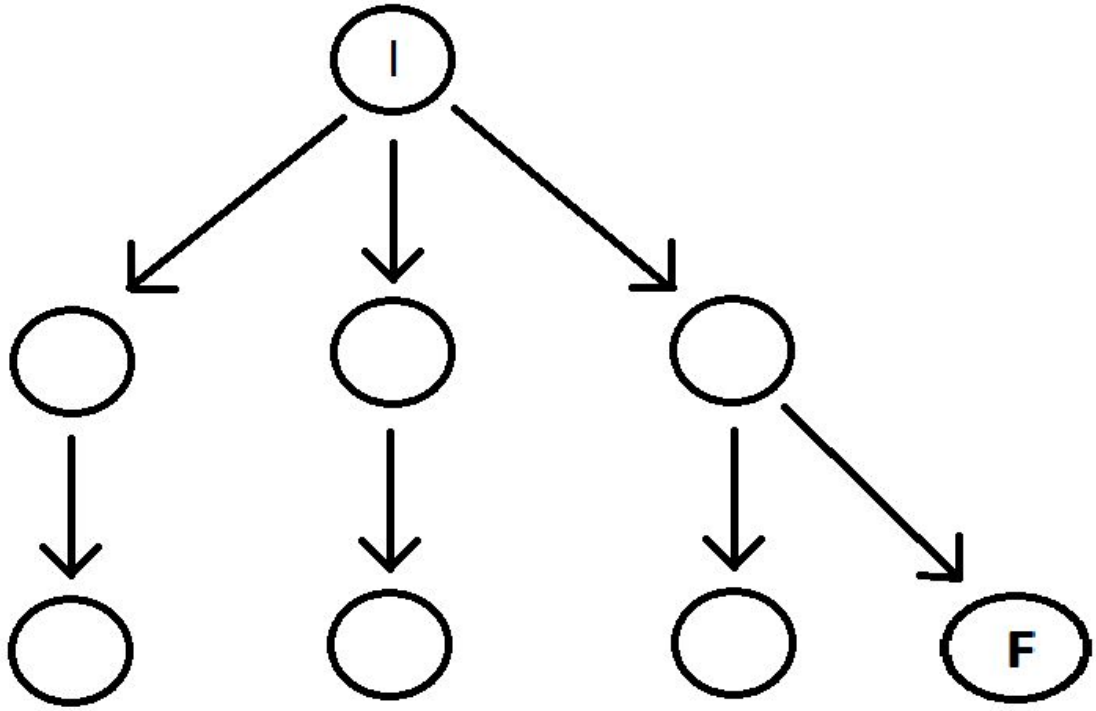
Mejor caso:
De igual forma, se tiene tiene que el caso más favorable es cuando el nodo final se encuentra en el primer nivel, siendo el óptimo, el primer nodo que recorre en el primer nivel.



(Caso 3)

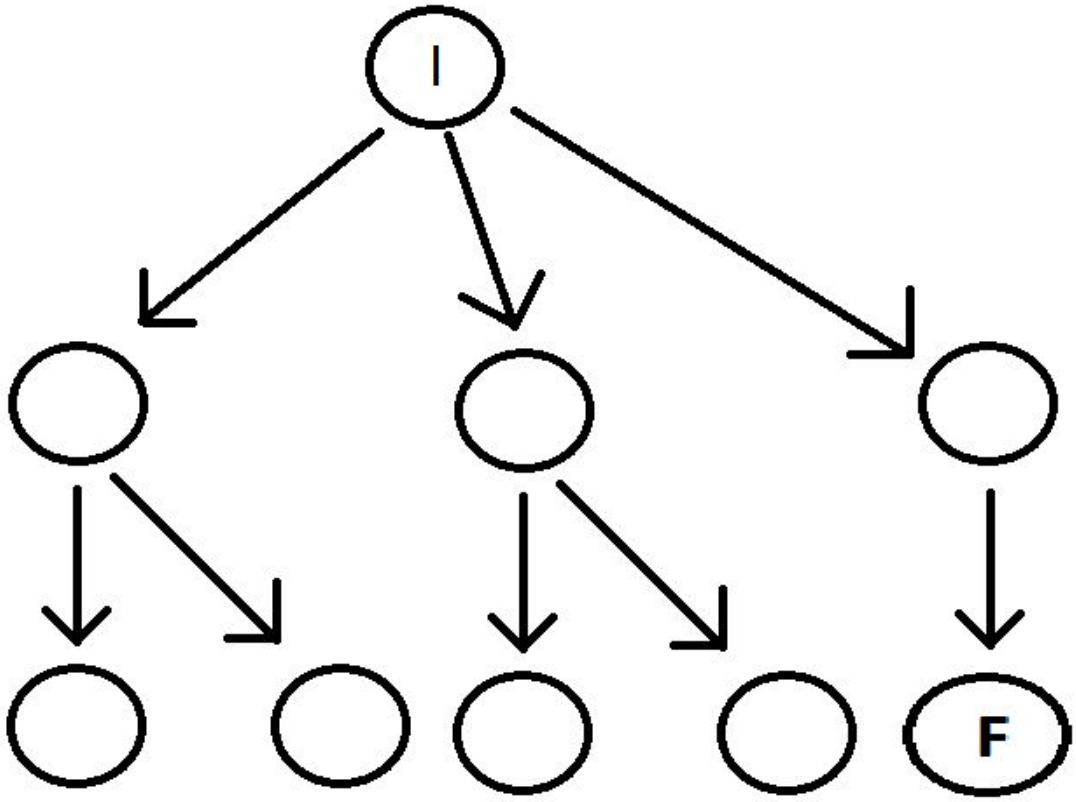
En el caso del BFS con heurística, la misma privilegia a aquellos nodos que cuentan con menos cantidad de hijos. De esta forma, se evita “over-headear” a aquellos nodos que son únicos hijos.

De esta forma el peor caso se encuentra cuando el nodo final es un nodo con muchos hermanos.



(Caso 4)

Y siendo el mejor caso, cuando es hijo único.



(Caso 5)

En código, los casos serían:

Caso 1, con 10 nodos:

```
caso1 = Digraph(10)

caso1.add_edge(0,1)
caso1.add_edge(1,2)
caso1.add_edge(2,3)
caso1.add_edge(3,4)
caso1.add_edge(4,5)
caso1.add_edge(5,6)
caso1.add_edge(6,7)
caso1.add_edge(7,8)
caso1.add_edge(8,9)

BFS(caso1,caso1.get_V(0),caso1.get_V(9))
BFS_heuristica(caso1,caso1.get_V(0),caso1.get_V(9))
```

Ambos casos harán: 0 > 1 > 2 > 3 > 4 > 5 > 6 > 7 > 8 > 9.

Caso 2, con 8 nodos:

```
caso2 = Digraph(8)

caso2.add_edge(0,1)
caso2.add_edge(0,2)
caso2.add_edge(0,3)
caso2.add_edge(1,4)
caso2.add_edge(2,5)
caso2.add_edge(3,6)
caso2.add_edge(4,7)

BFS(caso2,caso2.get_V(0),caso2.get_V(7))
BFS_heuristica(caso2,caso2.get_V(0),caso2.get_V(7))
```

Ambos casos harán: 0 > 1 > 2 > 3 > 4 > 5 > 6 > 7.

Caso 3, con 7 nodos:

```
caso3 = Digraph(7)

caso3.add_edge(0,1)
caso3.add_edge(0,2)
caso3.add_edge(0,3)
caso3.add_edge(1,4)
caso3.add_edge(2,5)
caso3.add_edge(3,6)
```

BFS(caso3,caso3.get_V(0),caso3.get_V(1))
BFS_heuristica(caso3,caso3.get_V(0),caso3.get_V(1))

Ambos casos harán: 0 > 1.

Caso 4, con 8 nodos:
caso4 = Digraph(8)

caso4.add_edge(0,1)
caso4.add_edge(0,2)
caso4.add_edge(0,3)
caso4.add_edge(1,4)
caso4.add_edge(2,5)
caso4.add_edge(3,6)
caso4.add_edge(3,7)

BFS_heuristica(caso4,caso4.get_V(0),caso4.get_V(7))

Ambos casos harán: 0 > 1 > 2 > 3 > 4 > 5 > 6 > 7.

Caso 5, con 9 nodos:
caso5 = Digraph(9)

caso5.add_edge(0,1)
caso5.add_edge(0,2)
caso5.add_edge(0,3)
caso5.add_edge(1,4)
caso5.add_edge(1,5)
caso5.add_edge(2,6)
caso5.add_edge(2,7)
caso5.add_edge(3,8)

BFS_heuristica(caso5,caso5.get_V(0),caso5.get_V(8))

Ambos casos harán: 0 > 1 > 2 > 3 > 8.

NOTA:
Las heurísticas que se pueden usar en este algoritmo están restringidas a la forma de recorrer el grafo que tiene BFS. Una de las principales importancias que tiene recorrer de esta forma un grafo es que la primera ocurrencia del nodo final siempre va a ser la de distancia mínima.
Por ende cambiar la heurística repercute en la naturaleza del mismo.
Lo único que se puede alterar es la prioridad con la que se saca de la “cola de prioridad”.
Por ende, cualquier heurística que se tome repercutirá en casos peores y mejores que tendrán similar forma (teniendo en cuenta que no se desea iterar por demás).

Complejidad de BFS:

```
def BFS(grafo,i,f):
        # recorremos todos los vértices del grafo inicializándolos a NO_VISITADO,
        # distancia INFINITA y padre de cada nodo NULL
        grafo.sacar_visitados()
        grafo.poner_distancias_inf()
        grafo.eliminar_padres()

        i.visitar()
        i.set_distancia(0)
        colaQ = []
        colaQ.append(i)

        termine = False
        if i.get_id() == f.get_id():
                termine = True

        while colaQ and not termine:
        # extraemos el nodo u de la cola Q y exploramos todos sus nodos adyacentes
                u = colaQ.pop(0)
                adj_u = grafo.adj(u.get_id())
                for v in adj_u:
                        if not v.fue_visitado():
                                v.visitar()
                                v.set_distancia(u.get_distancia() + 1)
                                v.set_padre(u)

                                colaQ.append(v)
                                if v.get_id() == f.get_id():
                                        return v.get_distancia()

        if not colaQ:
                return POSITIVE_INFINITY
```

Como se puede notar, en el ciclo principal (while), se recorren todos los nodos del grafo alcanzables desde el nodo inicial. A su vez, recorre todas las adyacencias del nodo en el segundo ciclo (for).
Por lo tanto, una cota del algoritmo siendo |vertices| = n y |elementos adyacentes| = m es: O(n+m).

Complejidad de BFS_heuristica:

```
def BFS_heuristica(grafo,i,f):
        # recorremos todos los vértices del grafo inicializándolos a NO_VISITADO,
        # distancia INFINITA y padre de cada nodo NULL

        grafo.sacar_visitados()
        grafo.poner_distancias_inf()
        grafo.eliminar_padres()

        i.visitar()
```

```
i.set_distancia(0)
cola = []
cola.append(i)

termine = False
if i.get_id() == f.get_id():
        termine = True

while cola and not termine:
        # extraemos el nodo u de la cola Q y exploramos todos sus nodos adyacentes
        u = cola.pop(0)
        adj_u = grafo.adj(u.get_id())
        cola_hijos_prior = Cola_prioridad()
        for v in adj_u:
                if not v.fue_visitado():
                        v.visitar()
                        v.set_distancia(u.get_distancia() + 1)
                        v.set_padre(u)
                        cola_hijos_prior.add_elem(grafo,v)
                        if v.get_id() == f.get_id():
                                return v.get_distancia()

        while not cola_hijos_prior.is_empty():
                elem = cola_hijos_prior.get_elem()
                cola.append(elem)

if not cola:
        return POSITIVE_INFINITY
```

Como se puede ver en la implementaciòn del BFS con heurística, el formato del algoritmo es casi igual al de BFS. Se debe notar que por la implementaciòn utilizada se agrega un ciclo de màs por la implementaciòn de la lista. Esto se puede optimizar dependiendo cómo se elija trabajarla y la finalidad de la misma.
Finalmente se concluye que al igual que BFS, el orden es de O(n+m).

Dijkstra

Complejidad:

```
def __init__(self, g,vo,vd):
        super(Dijkstra, self).__init__(g,vo,vd)
        heap = []
        heappush(heap,(0,vo))

        while heap and not g.get_V(vd).fue_visitado():
                verticeVisitado = heapq.heappop(heap)
                g.get_V(verticeVisitado[1]).visitar()
                verticesAdyacentes = g.adj(verticeVisitado[1])

                for vertice in verticesAdyacentes:
                        pesoDelVisitadoAlVertice = g.get_A(verticeVisitado[1], vertice.get_id()).get_weight()
                        if ( not vertice.fue_visitado() ) and ( vertice.get_distancia() > pesoDelVisitadoAlVertice):
                                vertice.set_distancia(pesoDelVisitadoAlVertice)
                                vertice.set_padre(verticeVisitado[1])
                                heappush(heap,(pesoDelVisitadoAlVertice,vertice.get_id()))
        self.armarResultado(vd)
```

Se puede ver que en cada iteración del ciclo while se obtiene de la cola de prioridad el vértice cuyo camino tiene el menor peso, obteniendo una complejidad de log(V), dentro del while, podemos ver un ciclo for, donde se recorrerán todos los vértices adyacentes, , en el cual cada vértice se podría conectar con los n-1 vértices restantes, es decir recorrer todas las aristas, teniendo una complejidad de A. Finalmente tendremos una complejida O(A . log(V)), Siendo A = aristas, V= vértices

A*

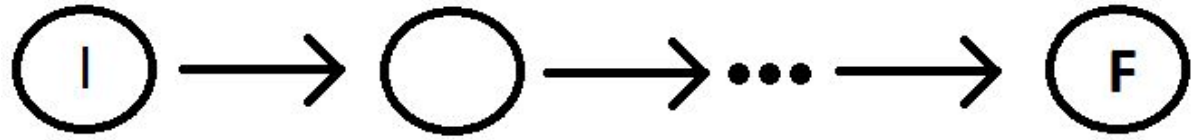
Complejidad:

```
def __init__(self,g,vo,vd):
        super(AEstrella, self).__init__(g,vo,vd)
        abiertos = []
        unCamino = [vo]
        par = [0,unCamino] #dejo f(x) como 0
        heappush(abiertos,par)
        final = False
        padre = None
        verticeActual = None

        while abiertos and not final:
                par = heapq.heappop(abiertos)
                idVerticeActual = par[1][-1]
                if idVerticeActual != vd:
                        vecinos = g.adj(idVerticeActual)
                        for v in vecinos:
                                id = v.get_id()
                                if (id != idVerticeActual):
                                        parAGuardar = [None,None]
                                        unCamino = par[1] + [id]
                                        parAGuardar = [__f__(unCamino,vd,g),unCamino]
                                        g.get_V(id).set_padre(idVerticeActual)
                                        heappush(abiertos,parAGuardar)


                        else:
                                final = True

                ###ARMO EL RECORRIDO
                if final:
                        self.resultado = par[1]
```

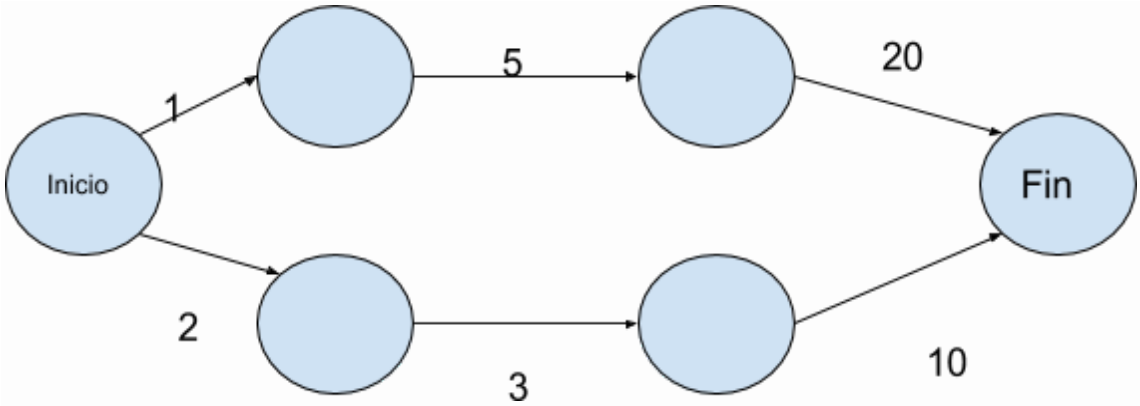
La complejidad del algoritmo A* se relaciona con su heurística, al usar una constante, este se transforma en un uniform cost search, teniendo una complejidad de $O(R^{d})$ siendo R el factor de ramificación (promedio de aristas que salen de un nod) y d la distancia de niveles del nodo inicial al nodo final.

Dijkstra y A*:

Un peor caso, al igual que los anteriores casos, será el de tener un grafo que funciona como lista:

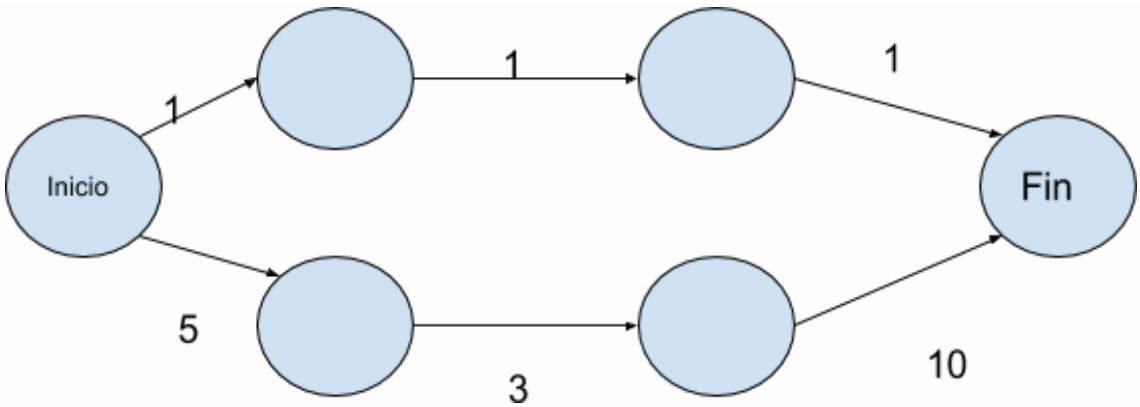


Para Dijkstra otro peor caso será en el que el camino mínimo candidato vaya cambiando todo el tiempo, por ej.:



Esto nos obligará a recorrer todos los nodos del grafo

Asimismo, el mejor caso será el de tener un camino mínimo constante:



El algoritmo A* depende mucho de su heurística, siendo el mejor caso para este algoritmo en el que la heurística sea igual al costo real de llegar al final h*(v) = r(v),

en los casos que h*(v) << r(v), solo sumará el g(v), y por tanto el algoritmo se comportará como un Dijkstra, guiándose solo por la cercanía de los nodos adyacentes, aunque se volverá mas lento ya que “expandirá” mas nodos. En los casos en que h*(v) > r(v) el algoritmo será rapido, pero no garantiza encontrar el camino mas corto. Finalmente, En los casos en que h*(v) >> r(v) , el A* se comportará como un Best-First-Search

estadisticos.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
###                          ###
###  ESTADISTICO DE ORDEN K  ###
###                          ###

# En ambas formas, cand es el indice en conj del numero candidato
# Asumiendo minimo k cuando hay repetidos
def fuerza_bruta_min_k(conj, cand, k):
    real_k = 1
    for i in conj:
        if (i < conj[cand]):
            real_k += 1
            if (real_k > k):
                return False
    if (real_k < k):
        return False
    return True

# Sin esa restriccion
def fuerza_bruta(conj, cand, k):
    min_real_k = 1
    diff_max_k = 0

    for idx, i in enumerate(conj):
        if (i < conj[cand]):
            min_real_k += 1
            if (min_real_k > k):
                return False
        elif (i == conj[cand] and idx != cand):
            diff_max_k += 1
    if (min_real_k + diff_max_k < k):
        return False
    return True

# Devuelve el estadístico orden-k como el resto de las funciones
def k_fuerza_bruta(conj, k):
    for idx_cand, cand in enumerate(conj):
        if fuerza_bruta(conj, idx_cand, k):
            return cand


# Usa el sort de Python
def ordenar_y_seleccionar(conj, k):
    conj.sort()
    return conj[k-1]


## K SELECCIONES
# k > 0
def k_selecciones(conj, k):
    for i in range(0, k):
        min = i
        for j in range(i+1,len(conj)):
            if conj[j] < conj[min]:
                min = j
        if i != min:
            conj[i] , conj[min] = conj[min] , conj[i]
    return conj[k-1]
```

```
# Para min-heapificar
def moveDownMin(conj, actual, last):
    hijo = 2 * actual + 1
    while (hijo <= last):
        if (hijo < last and conj[hijo] > conj[hijo + 1]):
            hijo += 1
        if (conj[actual] > conj[hijo]):
            conj[actual], conj[hijo] = conj[hijo], conj[actual]
            actual = hijo
            hijo = 2 * actual + 1
        else:
            return

def k_heapsort(conj, k):
    last = len(conj)
    for i in xrange(1, k+1):
        last -= 1
        # min-heapifico
        for j in xrange(last // 2, -1, -1):
            moveDownMin(conj, j, last)
        if i != k:
            conj[0], conj[last] = conj[last], conj[0]
    return conj[0]


# Para max-heapificar
def moveDownMax(conj, actual, last):
    hijo = 2 * actual + 1
    while (hijo <= last):
        if (hijo < last and conj[hijo] < conj[hijo + 1]):
            hijo += 1
        if (conj[actual] < conj[hijo]):
            conj[actual], conj[hijo] = conj[hijo], conj[actual]
            actual = hijo
            hijo = 2 * actual + 1
        else:
            return

# Para heap de máximo
def siftUpMax(conj, idx_actual):
    if idx_actual == 0:
        return
    idx_padre = idx_actual // 2
    if conj[idx_actual] > conj[idx_padre]:
        conj[idx_padre], conj[idx_actual] = conj[idx_actual], conj[idx_padre]
        siftUpMax(conj, idx_padre)

def heapselect(conj, k):
    maxHeap = []
    for idx, i in enumerate(conj):
        if (len(maxHeap) < k):
            maxHeap.append(i)
            siftUpMax(maxHeap, len(maxHeap)-1)
        elif (i < maxHeap[0]):
            maxHeap[0] = i
            moveDownMax(maxHeap, 0, len(maxHeap)-1)
    return maxHeap[0]
```

```
## QUICKSELECT
# k > 0
# Función para encontrar pivote y luego particionar el arreglo
def particionar(conj, ini, fin):
    i = ini - 1
    for j in range(ini, fin):
        if (conj[j] < conj[fin]):
            i += 1
            if (i != j):
                conj[i], conj[j] = conj[j], conj[i]
    # i+1 se va a convertir en el pivote (swap)
    conj[i+1], conj[fin] = conj[fin], conj[i+1]
    return i + 1

def quickselect(conj, ini, fin, k):
    k_real = k - 1
    if (ini == fin):
        return conj[k_real]
    # descartamos el subconj que no contiene al elemento k
    pivote = particionar(conj, ini, fin)
    if (k_real == pivote):
        return conj[k_real]
    if (k_real > pivote):
        ini = pivote + 1
    else:
        fin = pivote - 1
    return quickselect(conj, ini, fin, k)

# Devuelve el estadístico orden-k como el resto de las funciones
def k_quickselect(conj, k):
    return quickselect(conj, 0, len(conj)-1, k)
```

---

tests.py

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
from estadisticos import *

### TESTS ###

## fuerza_bruta
my_list = [4, 1, 5, 9, 12, 3, 0, 2, 7, 6]
true_k  = [5, 2, 6, 9, 10, 4, 1, 3, 8, 7]
for idx, k in enumerate(true_k):
        assert fuerza_bruta_min_k(my_list, idx, k)

my_list = [1, 1, 5, 12, 12, 5, 7, 2, 7, 6]
true_k  = [1, 2, 4,  9, 10, 5, 7, 3, 8, 6]
for idx, k in enumerate(true_k):
    assert fuerza_bruta(my_list, idx, k)

my_list  = [1, 1, 5, 12, 12, 5, 7, 2, 7, 6]
sorted_l = [1, 1, 2, 5, 5, 6, 7, 7, 12, 12]
for idx, i in enumerate(sorted_l):
    assert k_fuerza_bruta(my_list, idx+1) == i

## ordenar_y_seleccionar
my_list = [1, 1, 5, 12, 12, 5, 7, 2, 7, 6]
sorted_l = my_list
sorted_l.sort()
result = []
for k in xrange(1, 11):
    result.append(ordenar_y_seleccionar(my_list, k))
assert result == sorted_l

## k_selecciones
my_list  = [4, 1, 5, 9, 12, 3, 0, 2, 7, 6]
sorted_l = [0, 1, 2, 3, 4, 5, 6, 7, 9, 12]
for idx in xrange(0, 8):
    assert k_selecciones(my_list, idx+1) == idx

## k_heapsort
my_list  = [4, 1, 5, 9, 12, 3, 0, 2, 7, 6]
sorted_l = [0, 1, 2, 3, 4, 5, 6, 7, 9, 12]
for idx, i in enumerate(sorted_l):
    assert k_heapsort(my_list, idx+1) == i

## heapselect
my_list  = [4, 1, 5, 9, 12, 3, 0, 2, 7, 6]
sorted_l = [0, 1, 2, 3, 4, 5, 6, 7, 9, 12]
for idx, i in enumerate(sorted_l):
    assert heapselect(my_list, idx+1) == i

## quickselect
my_list  = [4, 1, 5, 9, 12, 3, 0, 2, 7, 6]
sorted_l = [0, 1, 2, 3, 4, 5, 6, 7, 9, 12]
for idx, i in enumerate(sorted_l):
    assert k_quickselect(my_list,  idx+1) == i
```

tadgrafo.py

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
import random
from tadCaminos import *
from constantes import *

class Vert:
    def __init__(self,node):
        self.id = node
        self.vecinos = {}
        self.visitado = False
        self.distancia = POSITIVE_INFINITY
        self.padre = None

    def add_neighbor (self ,dst,w = 0):
        self.vecinos[dst] = w

    def get_neighbors (self):
        return self.vecinos

    def get_id(self):
        return self.id

    def get_weight (self, dst):
        return self.vecinos[dst].get_weight()

    def visitar(self):
        self.visitado = True

    def fue_visitado(self):
        return self.visitado

    def setear_no_visitado(self):
        self.visitado = False

    def set_padre(self, padre):
        self.padre = padre

    def set_distancia(self, distancia):
        self.distancia = distancia

    def set_distancia_inf(self):
        self.distancia = POSITIVE_INFINITY

    def set_padre_vacio(self):
        self.padre = None

    def get_distancia(self):
        return self.distancia

    def get_padre(self):
        return self.padre


class Digraph:
    """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):  # @NoSelf
    """Construye un grafo sin aristas de V vértices.
    """
    g.aristas = []
    g.vertices = {}
    for i in range(0,V):
        new_vert = Vert(i)
        g.vertices[i] = new_vert

def V(g):   # @NoSelf
    """Número de vértices en el grafo.
    """
    return len(g.vertices)

def E(g):   # @NoSelf
    """Número de aristas en el grafo.
    """
    return len(g.aristas)

def adj_e(g, v):    # @NoSelf
    """Itera sobre los aristas incidentes _desde_ v.
    """
    aristas_incidentes_de_v = []
    for i in range(0,len(g.aristas)):
        if g.aristas[i].get_from() == v :
            aristas_incidentes_de_v.append(g.aristas[i])
    return aristas_incidentes_de_v



def adj(g, v):  # @NoSelf
    """Itera sobre los vértices adyacentes a ‘v’.
    """
    return g.vertices[v].get_neighbors().keys()


def add_edge(g, u, v, weight=0):    # @NoSelf
    """Añade una arista al grafo.
    """
    g.vertices[u].add_neighbor(g.vertices[v],weight)
    new_edge = Arista(u,v,weight)
    g.aristas.append(new_edge)


def __iter__(g):    # @NoSelf
    """Itera de 0 a V."""
    return g.vertices

def get_V(g,v): # @NoSelf
    return g.vertices[v]

def get_A(self, src, dst):
    for i in range(0,len(self.aristas)):
        arista = self.aristas[i]
        if (arista.get_from() == src ) and (arista.get_to() == dst) :
            return arista
    return None
```

```
    def iter_edges(g):  # @NoSelf
        """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(g.aristas)

    def sacar_visitados(g): # @NoSelf
        for i in range(0,len(g.vertices)):
            g.vertices[i].setear_no_visitado()

    def poner_distancias_inf(g): # @NoSelf
        for i in range(0,len(g.vertices)):
            g.vertices[i].set_distancia_inf()

    def eliminar_padres(g): # @NoSelf
        for i in range(0,len(g.vertices)):
            g.vertices[i].set_padre_vacio()


class Arista:
    """Arista de un grafo.
      """
    def __init__(self, src, dst, weight=0):
        # inicializar y do things
        self.src = src
        self.dst = dst
        self.weight = weight

    def get_weight (self):
        return self.weight

    def get_from (self):
        return self.src

    def get_to (self):
        return self.dst
```

---

tadCaminos.py

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
from abc import ABCMeta, abstractmethod
import heapq
from tadgrafo import *
from _heapq import heappush
from constantes import *

class Caminos():
	__metaclass__ = ABCMeta

	@abstractmethod
	def __init__(self, g,vo,vd, heuristica = None):
		self.resultado = []
		self.g = g
		pass

	def distancia(self, v):
		if v not in self.resultado: return POSITIVE_INFINITY
		total = 0
		for i in range(0,len(self.resultado)):
			if self.resultado[i] == v:
				return total
			total = total+self.g.get_A(self.resultado[i],self.resultado[i+1]).get_weight()

	def visitado(self, v):
		for i in range(0,len(self.resultado)):
			if self.resultado[i] == v:
				return True
		return False

	def camino(self):
		return self.resultado



class Dijkstra(Caminos):

	def __init__(self, g,vo,vd,heuristica = None):
		super(Dijkstra, self).__init__(g,vo,vd,heuristica = None)
		heap = []
		heappush(heap,(0,vo))

		while heap and not g.get_V(vd).fue_visitado():
			verticeVisitado = heapq.heappop(heap)
			g.get_V(verticeVisitado[1]).visitar()
			verticesAdyacentes = g.adj(verticeVisitado[1])

			for vertice in verticesAdyacentes:
				pesoDelVisitadoAlVertice = g.get_A(verticeVisitado[1],
vertice.get_id()).get_weight()
				if ( not vertice.fue_visitado() ) and ( vertice.get_distancia() >
pesoDelVisitadoAlVertice):
					vertice.set_distancia(pesoDelVisitadoAlVertice)
					vertice.set_padre(verticeVisitado[1])
					heappush(heap,(pesoDelVisitadoAlVertice,vertice.get_id()))
		self.armarResultado(vd)

	def armarResultado(self,vd):
```

```
                self.resultado = [vd]
                padre = self.g.get_V(vd).get_padre()
                while padre != None:
                        self.resultado = [padre] + self.resultado
                        padre = self.g.get_V(padre).get_padre()



class AEstrella(Caminos):

        def __init__(self,g,vo,vd,heuristica = None):
                super(AEstrella, self).__init__(g,vo,vd,heuristica = None)
                abiertos = []
                unCamino = [vo]
                par = [0,unCamino] #dejo f(x) como 0
                heappush(abiertos,par)
                final = False
                padre = None
                verticeActual = None

                while abiertos and not final:
                        par = heapq.heappop(abiertos)
                        idVerticeActual = par[1][-1]
                        if idVerticeActual != vd:
                                vecinos = g.adj(idVerticeActual)
                                for v in vecinos:
                                        id = v.get_id()
                                        if (id != idVerticeActual):
                                                parAGuardar = [None,None]
                                                unCamino = par[1] + [id]
                                                parAGuardar = [__f__(unCamino,vd,g,heuristica),unCamino]
                                                g.get_V(id).set_padre(idVerticeActual)
                                                heappush(abiertos,parAGuardar)


                        else:
                                final = True

                ###ARMO EL RECORRIDO
                if final:
                        self.resultado = par[1]

def __f__(camino, vd, g,heuristica):
        return __g__(camino, g) + heuristica(g,camino[-1])

def __g__(camino, g):
        total = 0
        for elem in range(0,len(camino)-1):
                total = total + g.get_A(camino[elem], camino[elem+1]).get_weight() #sumo los pesos de las
aristas
        return total

def __heuristica__(grafo,elem):
        adj_elem = grafo.adj(elem)
        cant = 1
        for elem in adj_elem:
                if(not elem.fue_visitado()):
                        cant = cant + 1
        return cant
```

---

bfs.py

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
from tadgrafo import *

def armarResultado(grafo,vd):
	resultado = [vd]
	padre = grafo.get_V(vd).get_padre()
	while padre != None:
		resultado = [padre.get_id()] + resultado
		padre = grafo.get_V(padre.get_id()).get_padre()
	return resultado


def BFS(grafo,inf,fin):
	# recorremos todos los vértices del grafo inicializándolos a NO_VISITADO,
	# distancia INFINITA y padre de cada nodo NULL
	i = grafo.get_V(inf)
	f = grafo.get_V(fin)

	grafo.sacar_visitados()
	grafo.poner_distancias_inf()
	grafo.eliminar_padres()

	i.visitar()
	i.set_distancia(0)
	colaQ = []
	colaQ.append(i)

	termine = False
	if i.get_id() == f.get_id():
		termine = True

	while colaQ and not termine:
	# extraemos el nodo u de la cola Q y exploramos todos sus nodos adyacentes
		u = colaQ.pop(0)
		adj_u = grafo.adj(u.get_id())
		for v in adj_u:
			if not v.fue_visitado():
				v.visitar()
				v.set_distancia(u.get_distancia() + 1)
				v.set_padre(u)

				colaQ.append(v)
				if v.get_id() == f.get_id():
					return v.get_distancia()

	if not colaQ:
		return POSITIVE_INFINITY
	else: return "Error, algo salio muy mal..."
```

busqueda_heuristica.py

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
from tadgrafo import *

def __heuristica__(grafo,elem):
	adj_elem = grafo.adj(elem.get_id())
	cant = 1
	for elem in adj_elem:
		if(not elem.fue_visitado()):
			cant = cant + 1
	return cant

def BFS_heuristica(grafo,inf,fin):
	# recorremos todos los vértices del grafo inicializándolos a NO_VISITADO,
	# distancia INFINITA y padre de cada nodo NULL
	i = grafo.get_V(inf)
	f = grafo.get_V(fin)

	grafo.sacar_visitados()
	grafo.poner_distancias_inf()
	grafo.eliminar_padres()

	i.visitar()
	i.set_distancia(0)
	cola = []
	cola.append(i)

	termine = False
	if i.get_id() == f.get_id():
		termine = True

	while cola and not termine:
		# extraemos el nodo u de la cola Q y exploramos todos sus nodos adyacentes
		u = cola.pop(0)
		adj_u = grafo.adj(u.get_id())
		cola_hijos_prior = Cola_prioridad()
		for v in adj_u:
			if not v.fue_visitado():
				v.visitar()
				v.set_distancia(u.get_distancia() + 1)
				v.set_padre(u)
				cola_hijos_prior.add_elem(grafo,v)
				if v.get_id() == f.get_id():
					return v.get_distancia()

		while not cola_hijos_prior.is_empty():
			elem = cola_hijos_prior.get_elem()
			cola.append(elem)

	if not cola:
		return POSITIVE_INFINITY
	else: return "Error, algo salio muy mal..."

class Cola_prioridad:
	def __init__(self):
		self.lista = []

	def get_cola(self):
		return self.lista
```

```
        def is_empty(self):
                if not self.lista: return True
                else: return False

        def add_elem (self,grafo,elem):
                elem_prior = []
                prioridad = __heuristica__(grafo,elem)
                elem_prior.append(elem)
                elem_prior.append(prioridad)
                self.lista.append(elem_prior)


        def get_elem (self):
                max_prior = POSITIVE_INFINITY
                elem = None
                cont = 0
                ind = 0

                if not self.lista:
                        return None
                for elem_prior in self.lista:
                        if elem_prior[1] < max_prior:
                                max_prior = elem_prior[1]
                                elem = elem_prior[0]
                                ind = cont
                        cont = cont + 1
                self.lista.pop(ind)
                return elem


"""
graph = Digraph(19)
graph.add_edge( 0, 1, 1)
graph.add_edge( 0, 2, 2)
graph.add_edge( 0, 3, 3)
graph.add_edge( 1, 4, 4)
graph.add_edge( 1, 5, 5)
graph.add_edge( 2, 6, 6)
graph.add_edge( 2, 7, 7)
graph.add_edge( 2, 8, 8)
graph.add_edge( 3, 9, 9)
graph.add_edge( 4, 14, 10)
graph.add_edge( 5, 15, 11)
graph.add_edge( 6, 11, 12)
graph.add_edge( 7, 12, 13)
graph.add_edge( 8, 13, 14)
graph.add_edge( 9, 10, 15)
graph.add_edge( 10, 16, 16)

ads = BFS_heuristica(graph,graph.get_V(0),graph.get_V(18))
"""
```

---

constanstes.py

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
POSITIVE_INFINITY = float("inf")
```

main.py

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

from tadCaminos import *
from constantes import *

from tadgrafo import *
from busqueda_heuristica import *
from bfs import *

def mockHeuristica(g,v):
    #heuristica para probar el ejemplo de http://stackoverflow.com/questions/5849667/a-search-algorithm
        if v == 1: return 5
        if v == 2: return 6
        if v == 3: return 8
        if v == 4: return 5
        if v == 5: return 4
        if v == 6: return 15
        return 0

def mock2Heuristica(g,v):
    #heuristica para probar el ejemplo de
http://stackoverflow.com/questions/20162735/a-algorithm-on-a-directed-graph?noredirect=1&lq=1
    if v == 1: return 7
    if v == 2: return 0
    if v == 3: return 11
    if v == 4: return 5
    if v == 5: return 1
    if v == 0: return 8
    return 0


d4 = Digraph(4)

print "aristas = " + str(d4.E())
print "vertices = " + str(d4.V())

# testeo las funciones de Digraph
print "agrego aristas"
d4.add_edge( 1, 2, 20)
d4.add_edge( 0, 1, 3)
d4.add_edge( 0, 3, 15)
d4.add_edge( 1, 3, 9)
d4.add_edge( 1, 1)
d4.add_edge( 2, 3, 7)

print "aristas = " + str(d4.E())
print "vertices = " + str(d4.V())

print "testeo Dijkstra"
dijkstra1 = Dijkstra( d4, 0, 3)
print dijkstra1.camino()

print dijkstra1.distancia(0)
print dijkstra1.distancia(1)
print dijkstra1.distancia(2)
print dijkstra1.distancia(3)
print dijkstra1.visitado(0)
print dijkstra1.visitado(1)
```

```
print dijkstra1.visitado(2)
print dijkstra1.visitado(3)

print "otro Dijkstra"
d6 = Digraph(6)
d6.add_edge( 0, 1, 4)
d6.add_edge( 0, 2, 2)
d6.add_edge( 1, 3, 5)
d6.add_edge( 2, 3, 8)
d6.add_edge( 2, 4, 10)
d6.add_edge( 3, 4, 2)
d6.add_edge( 4, 3, 2)
d6.add_edge( 3, 5, 6)
d6.add_edge( 4, 5, 2)

dijkstra2 = Dijkstra( d6, 0, 5)
print dijkstra2.camino()
print dijkstra2.visitado(1)
print dijkstra2.distancia(4)

print "testeo A*"
"""http://stackoverflow.com/questions/5849667/a-search-algorithm
S=0
A=1
B=2
Y=3
X=4
C=5
D=6
E=7"""

d8 = Digraph(8)
d8.add_edge( 0, 1, 1)
d8.add_edge( 0, 2, 2)
d8.add_edge( 1, 3, 7)
d8.add_edge( 1, 4, 4)
d8.add_edge( 2, 5, 7)
d8.add_edge( 2, 6, 1)
d8.add_edge( 3, 7, 3)
d8.add_edge( 4, 7, 2)
d8.add_edge( 5, 7, 5)
d8.add_edge( 6, 7, 12)


Astar1 = AEstrella( d8, 0, 7,mockHeuristica)
print Astar1.camino()

""" ej de :
http://stackoverflow.com/questions/20162735/a-algorithm-on-a-directed-graph?noredirect=1&lq=1
"""

print "otro de A*"
d7 = Digraph(7)
d7.add_edge( 0, 1, 2)
d7.add_edge( 0, 2, 11)
d7.add_edge( 0, 3, 1)
d7.add_edge( 1, 4, 3)
d7.add_edge( 2, 1, 2)
d7.add_edge( 2, 6, 1)
d7.add_edge( 2, 5, 1)
d7.add_edge( 3, 5, 15)
d7.add_edge( 3, 2, 12)
```

```
d7.add_edge( 4, 2, 5)
d7.add_edge( 4, 6, 7)
d7.add_edge( 5, 6, 1)

AStar2 = AEstrella( d7, 0, 6,mock2Heuristica)
print AStar2.camino()
print AStar2.distancia(0)
print AStar2.distancia(4)
print AStar2.distancia(5)
print AStar2.visitado(0)
print AStar2.visitado(4)
print AStar2.visitado(5)

gbfs1 = Digraph(9)
gbfs1.add_edge( 0, 1)
gbfs1.add_edge( 0, 2)
gbfs1.add_edge( 0, 3)
gbfs1.add_edge( 1, 4)
gbfs1.add_edge( 1, 5)
gbfs1.add_edge( 2, 6)
gbfs1.add_edge( 3, 7)
gbfs1.add_edge( 3, 8)

ads = BFS_heuristica(gbfs1,0,6)

print "distancia al nodo final = "+str(ads)

gbfs2 = Digraph(19)
gbfs2.add_edge( 0, 1)
gbfs2.add_edge( 0, 2)
gbfs2.add_edge( 0, 3)
gbfs2.add_edge( 1, 4)
gbfs2.add_edge( 1, 5)
gbfs2.add_edge( 2, 6)
gbfs2.add_edge( 2, 7)
gbfs2.add_edge( 2, 8)
gbfs2.add_edge( 3, 9)
gbfs2.add_edge( 4, 14)
gbfs2.add_edge( 5, 15)
gbfs2.add_edge( 6, 11)
gbfs2.add_edge( 7, 12)
gbfs2.add_edge( 8, 13)
gbfs2.add_edge( 9, 10)
gbfs2.add_edge( 10, 16)

ads = BFS(gbfs2,0,16)
print "distancia al nodo final = "+str(ads)


asd = armarResultado(gbfs1,6)
print asd

asd = armarResultado(gbfs2,16)
print asd
```
