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. Estadístico de orden k

1.1. Brute Force

1.1.1. Complejidad

A simple vista, calcular si un elemento es o no, por ejemplo, el 4^{to} elemento más pequeño, es $O(n)$, ya que se fija cuántos son menores a él entre todos los demás elementos del conjunto. Como lo hacemos potencialmente para todos los elementos (ya que, en el peor caso, el *k-ésimo* elemento puede ser el último), este método es $O(n^2)$.

1.1.2. Mejor caso

Cuando el *k-ésimo* elemento es el primero en la lista.

Ejemplo:

$k = 4$

$l = [4, 15, 2, 1, 0, 14, 6, 11, 8, 9, 3, 13, 12, 7, 5, 10]$

$O(n)$: En este caso, al comenzar a comparar desde el primer elemento, encontramos el 'k' requerido en la primer iteración.

1.1.3. Peor caso

Cuando el *k-ésimo* elemento es el último en la lista.

Ejemplo:

$k = 4$

$l = [10, 15, 2, 1, 0, 14, 6, 11, 8, 9, 3, 13, 12, 7, 5, 4]$

$O(n^2)$: En este caso, encontramos el 'k' elegido en la última iteración (luego de recorrer todo el conjunto para cada uno de los elementos anteriores a 4).

1.2. Order and Select

1.2.1. Complejidad

Para el presente trabajo práctico se utilizó el Timsort (algoritmo de ordenamiento usado por `Python`), que es $O(n \log n)$ en el peor caso y $O(n)$ en el mejor caso. Una vez que se ordena el conjunto, se toma el elemento k de la lista en $O(1)$. Por lo tanto, este algoritmo sería $O(n \log n)$.

1.2.2. Mejor caso

El mejor caso del Timsort se da cuando la entrada ya está ordenada (ya sea en forma ascendente o descendente). Luego, encontrar el k-esimo elemento es constante.

Ejemplo:

$k = cualquiera$

$l = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]$

$O(n)$

1.2.3. Peor caso

Lamentablemente no pudimos encontrar un ejemplo claro y concreto para demostrar el peor caso del Timsort

$O(n \log n)$

1.3. k-selecciones

1.3.1. Complejidad

En la selección se analizan los n elementos de la lista y se pone en primer lugar al más pequeño. Luego, sobre los $n - 1$ restantes se repite el proceso, luego sobre $n - 2$ y así k veces en total. Cada una de las selecciones, sobre una lista de n elementos, es $O(n)$ (debe recorrerlos todos para ver el mínimo).

Este algoritmo tiene k selecciones, con lo cual es entonces $O(k * n)$ (tanto k como n son parte de la entrada, ninguna es constante). Ya que $k < n$, esto seguramente sea menor a $O(n^2)$. Salvo que k sea menor a $\log(n)$, este algoritmo es superado por el Order and Select.

1.3.2. Mejor caso

La complejidad del algoritmo depende de k, por lo que el mejor caso se da cuando $k = 0$ (o 1, dependiendo dicha notación si se usa 0-based o no), donde realiza la selección parcial 1 sola vez, y esto se da con cualquier entrada de cualquier tamaño n.

$k = 0$

$l = [4, 15, 2, 1, 0, 14, 6, 11, 8, 9, 3, 13, 12, 7, 5, 10]$

1.3.3. Peor caso

Por el mismo argumento anterior, el peor caso se da cuando $k = n$, ya que debe realizar las selecciones parciales de absolutamente todos los elementos del arreglo.

$k = 15$

$l = [4, 15, 2, 1, 0, 14, 6, 11, 8, 9, 3, 13, 12, 7, 5, 10]$

1.4. k-heapsort

1.4.1. Complejidad

Un heapsort es un ordenamiento de selección, sólo que se usa un heap de mínimo para obtener el mínimo en $O(\log n)$ en lugar de $O(n)$. Si bien observar el mínimo de un heap es $O(1)$, como lo quitaremos debe hacerse un upheap o float, que es $O(\log n)$.

Entonces, el heapsort consiste en dos etapas:

1. Heapify. Se reordena el array para que sea un heap. Esto es $O(n)$.
2. Las n extracciones del mínimo $O(n \log n)$.

Por lo tanto, un heapsort normalmente es $O(n) + O(n \log n) = O(n \log n)$.

Aquí, al igual que con el k-selecciones, quitaremos los primeros k elementos. Por esto, aquí se hará el heapify, sucedido de k extracciones del mínimo. El orden sería $O(n + k \log n)$ con $k < n$.

1.4.2. Mejor caso

Como en este caso se realiza el heapify sea cual sea la entrada (no importa el tamaño), el mejor o peor caso se da dependiendo de la cantidad de extracciones que deban hacerse. Es decir, depende de k. El mejor caso se da cuando hay que hacer una sola extracción, es decir, cuando $k = 0$

$l = [4, 15, 2, 1, 0, 14, 6, 11, 8, 9, 3, 13, 12, 7, 5, 10]$

1.4.3. Peor caso

Siguiendo el razonamiento anterior, el peor caso se da cuando $k = n - 1$ ya que hay que realizar n extracciones.

$l = [4, 15, 2, 1, 0, 14, 6, 11, 8, 9, 3, 13, 12, 7, 5, 10]$

1.5. HeapSelect

1.5.1. Complejidad

Este algoritmo consiste en lo siguiente:

1. Toma los primeros k elementos de la lista y los convierte en un heap de máximo con heapify. Esto es $O(k)$.
2. Mantiene los k mínimos en el heap. Para esto recorre cada elemento de la lista y, si es menor al máximo lo agrega al heap y quita el nuevo máximo. Si es mayor al máximo, no lo agrega. Para cada elemento cuesta $O(\log k)$ con lo cual este paso completo es $O(n \log k)$.
3. Observamos la raíz del heap final (Máximo de los k mínimos elementos) lo cual es $O(1)$.

El tiempo entonces es $O(k + n \log k + 1) = O(n \log k)$.

Esta función es rápida para k pequeño. Si k se acerca a n, termina siendo más conveniente directamente ordenar la lista, ya que de todos modos tendremos que ordenar en el 3er paso la mayor parte de los elementos en el paso 3. Esta comparación de cuándo conviene cada una es justamente una comparación entre este algoritmo y el Order and Select.

1.5.2. Mejor caso

Siendo que el paso número 1 es indiferente sea cual sea la entrada, n y k, y que la observación del máximo elemento del heap es $O(1)$ siempre, resta entender que para que se dé el mejor caso, nos conviene realizar la menor cantidad de inserciones en el heap, a medida que recorremos la lista de $n - k$ elementos restantes en el paso número 2.

Por ende, el mejor caso se da cuando la entrada está parcialmente ordenada, siendo los primeros k elementos, ya los mínimos de toda la entrada (lo cual equivaldría a decir que de los $n - k$ elementos restantes que se iteran en el punto 2, ninguno se insertaría en el heap).

$k = 4$

$l = [4, 3, 2, 1, 0, 14, 6, 11, 8, 9, 15, 13, 12, 7, 5, 10]$

Puede verse que los primeros $k = 4$ elementos (0-based) entrarían en el heap inicial, y los restantes $n - k = 12$ no entrarían en el heap a medida que se los itera (ya que ninguno es más pequeño que 4, la raíz del heap que se genera luego del punto 1.

Vale aclarar que existe otro mejor caso, el cual depende absolutamente de k y se da cuando $k = n - 1$, es decir, cuando se quiere encontrar el máximo elemento del conjunto. Este caso equivale a un HeapSort (ya que hay que ordenar en un heap absolutamente todos los elementos, el paso 2 no existiría, y leer de la raíz es constante), por lo que también se da en $O(n)$

1.5.3. Peor caso

Con el mismo razonamiento anterior, el peor caso se da cuando hay que realizar absolutamente todas las inserciones al heap generado en el punto 1. El caso se daría cuando, con un k determinado, los primeros k elementos son los mayores del conjunto (no importa que este sub-conjunto esté ordenado), y luego los restantes $n - k$ elementos estén ordenados en orden decreciente.

$k = 12$

$l = [4, 12, 7, 5, 10, 14, 6, 11, 8, 9, 15, 13, 3, 2, 1, 0]$

1.6. QuickSelect

1.6.1. Complejidad

Se usa la lógica del Quicksort: se define un pivote y se ponen todos los elementos mayores "a la derecha" y todos los menores "a la izquierda". Según la posición p del pivote:

- Si $p == k$: se devuelve `l[p]`
- Si $p < k$: se hace QuickSelect sobre la parte derecha (`l[p+1..n]`)
- Si $p > k$: se hace QuickSelect sobre la parte izquierda(`l[0..p]`)

Esto entonces, en el caso óptimo de que el pivote siempre queda a la mitad tiene un tiempo de $T(n) = T(n/2) + O(n)$ (n es lo que tarda en poner todo en su lado correspondiente del pivote).

Utilizando el Teorema Maestro, esto es $O(n)$.

Ahora bien, esto depende mucho de cómo se elige al pivote. Si se tiene una elección de pivote mala para lo que deseamos buscar, es factible ir particionando el arreglo de a 1 elemento, por lo que estaríamos en un caso de $O(n^2)$

1.6.2. Mejor caso

Como dijimos, depende mucho de la elección del pivote. Si, por ejemplo, elegimos el pivote que se encuentra en la primer posición del conjunto, se quiere buscar el primer mínimo (k == 0 en 0-based) y, además, el conjunto está ordenado, entonces lo encontraremos en $O(n)$, ya que el primer pivote utilizado es el que quiero devolver

$k = 0$

$l = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]$

1.6.3. Peor caso

Si por el contrario, y tomando como base el caso anterior, queremos el máximo elemento (es decir, el n-esimo elemento más pequeño), pero siempre elegimos al primer elemento como pivote y además el conjunto está ordenado en forma ascendente, llegamos al peor caso (ir particionando el conjunto de a 1 elemento)

$k = 15$

$l = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]$

1.7. Nota general sobre k

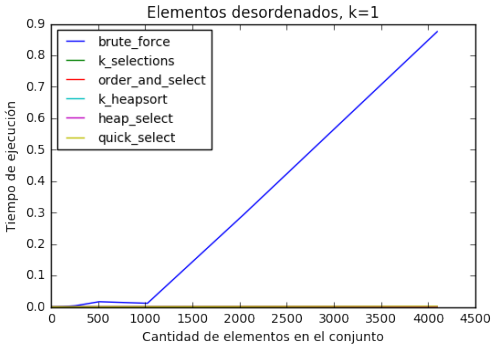
$k <= n$. Sería absurdo pedir algo como el $n + 1$ menor elemento.

Si $k == n$, entonces el k mínimo es el máximo, y puede encontrarse en $O(n)$ más rápido que con cualquiera de los otros. De hecho, todos los algoritmos antes descriptos pueden ser mejorados sustancialmente con este razonamiento. Si $k > n/2$, entonces el *k-mínimo* es el *n-k-máximo* y será más barato buscar al *k-máximo* con el mismo algoritmo.

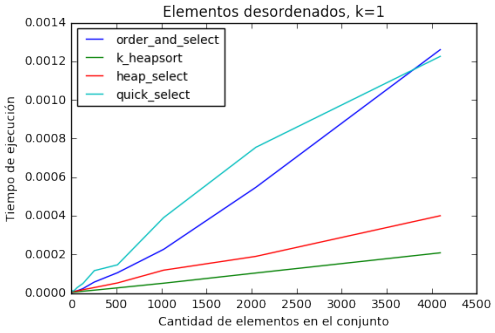
1.8. Comparación de tiempos de ejecución

Visualizaremos los tiempos de ejecución de cada algoritmo, variando el tamaño de la entrada y obteniendo resultados para buscar los casos particulares $k = 1$, $k = n/2$ y $k = n$.

1.8.1. $k = 1$

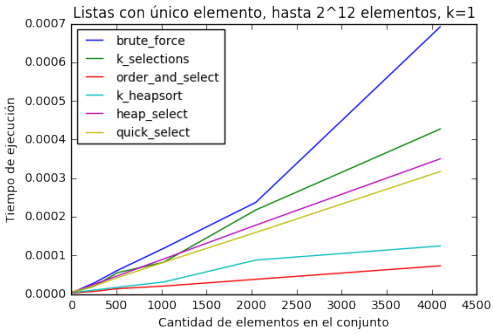


(a) Con algoritmos exponenciales

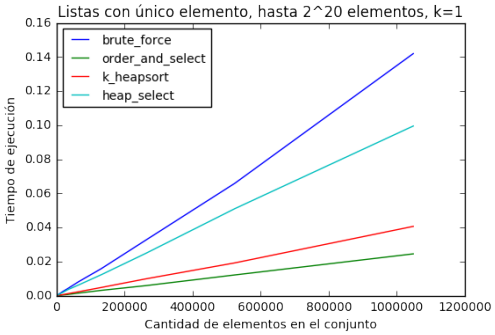


(b) Sin algoritmos exponenciales

Figura 1: $k = 1$, elementos desordenados

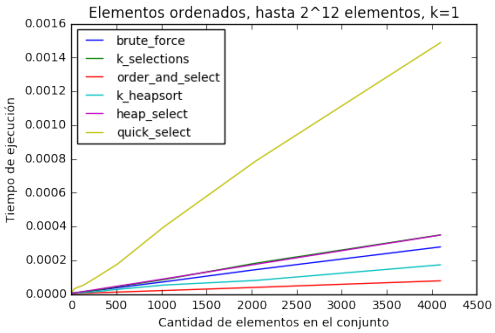


(a) Con algoritmos exponenciales

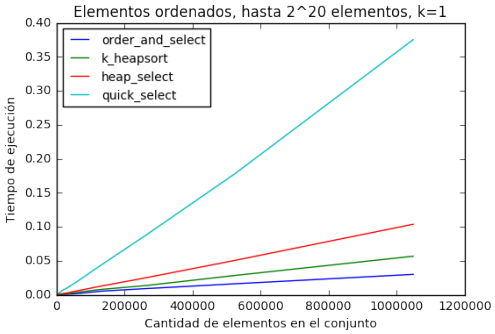


(b) Sin algoritmos exponenciales

Figura 2: $k = 1$, elemento único



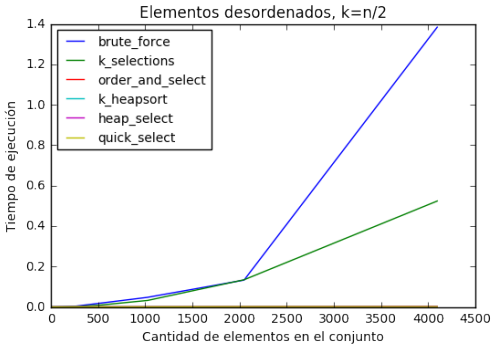
(a) Con algoritmos exponenciales



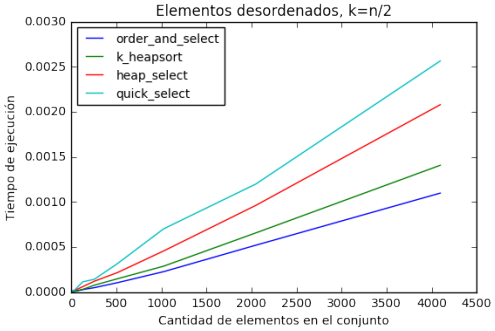
(b) Sin algoritmos exponenciales

Figura 3: $k = 1$, elementos ordenados

1.8.2. $k = n/2$

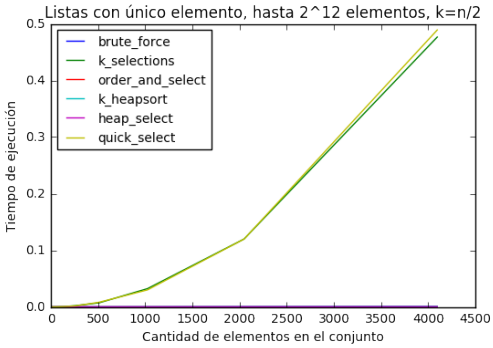


(a) Con algoritmos exponenciales

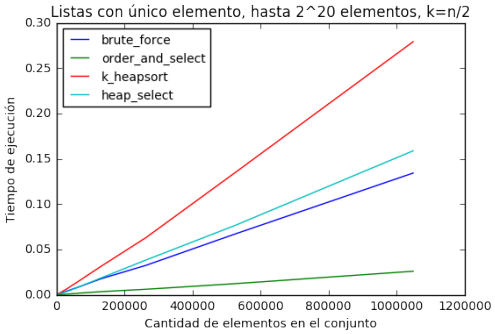


(b) Sin algoritmos exponenciales

Figura 4: $k = n/2$, elementos desordenados

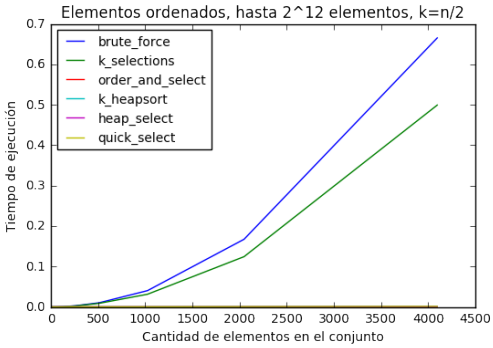


(a) Con algoritmos exponenciales

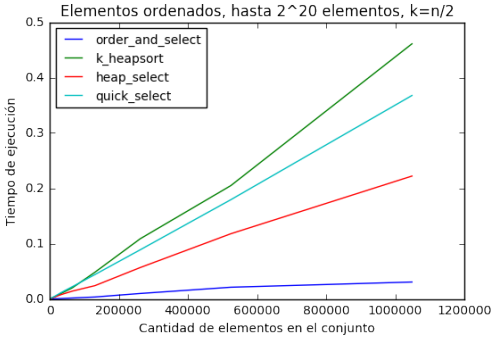


(b) Sin algoritmos exponenciales

Figura 5: $k = n/2$, elemento único



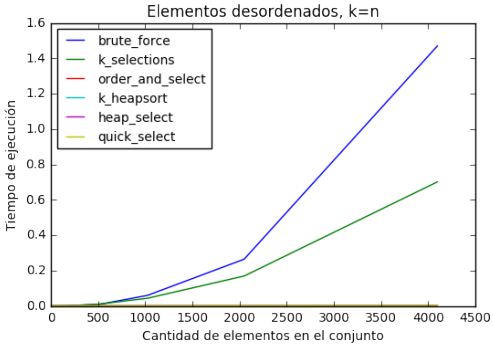
(a) Con algoritmos exponenciales



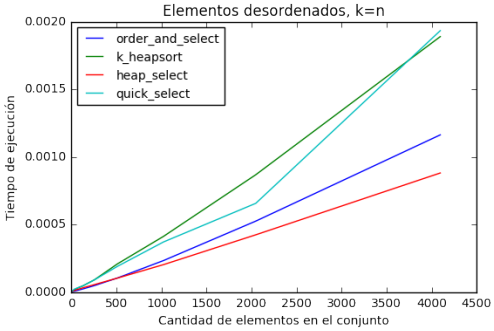
(b) Sin algoritmos exponenciales

Figura 6: $k = n/2$, elementos ordenados

1.8.3. $k = n$

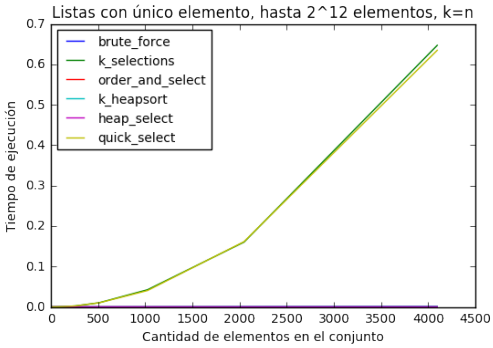


(a) Con algoritmos exponenciales

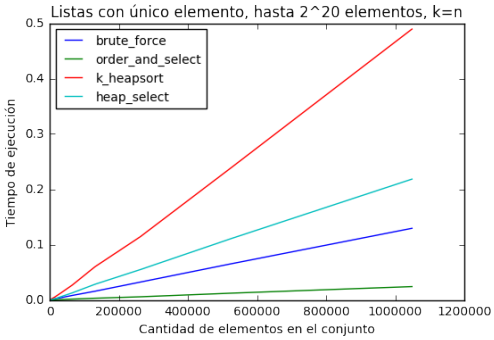


(b) Sin algoritmos exponenciales

Figura 7: $k = n$, elementos desordenados

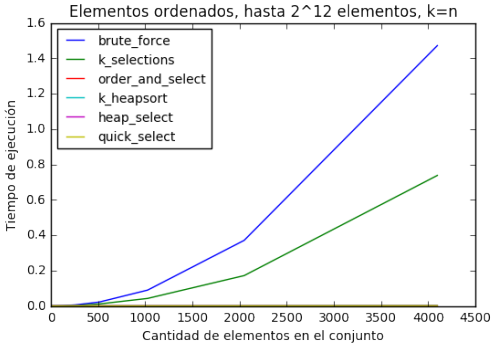


(a) Con algoritmos exponenciales

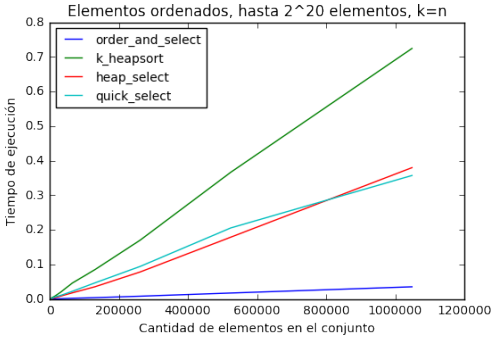


(b) Sin algoritmos exponenciales

Figura 8: $k = n$, elemento único



(a) Con algoritmos exponenciales



(b) Sin algoritmos exponenciales

Figura 9: $k = n$, elementos ordenados

1.9. Elección de algoritmo óptimo para cada k según n

Podemos ver una aproximación del cálculo cualitativo en el siguiente gráfico

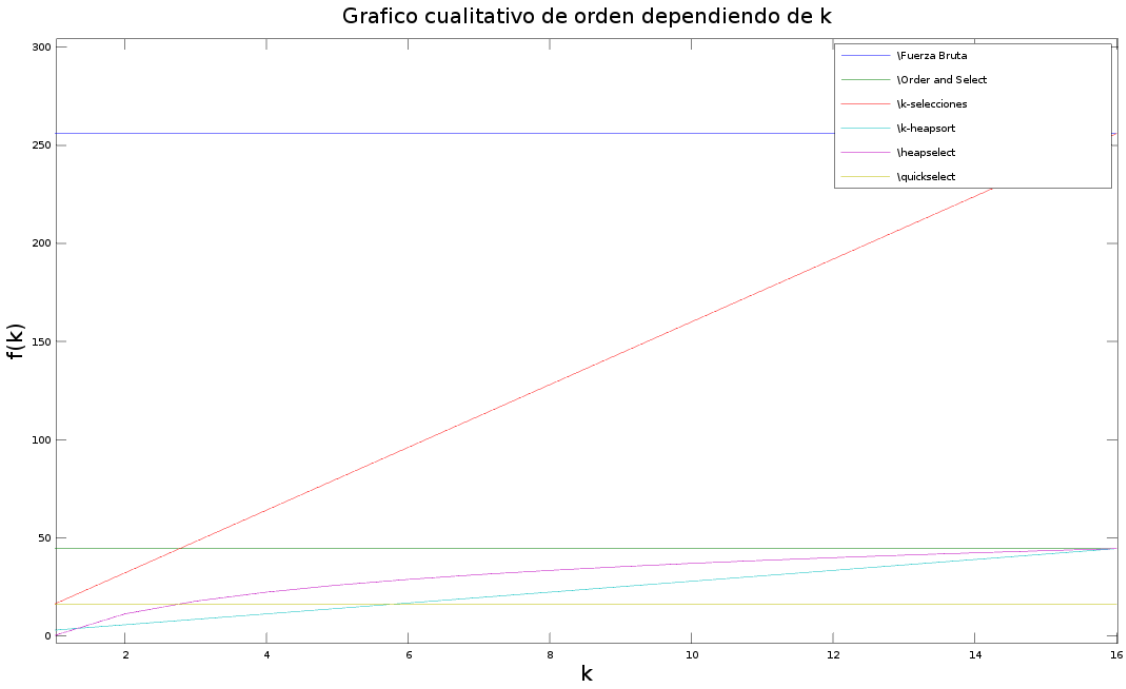


Figura 10: Comparación del k óptimo para todos los algoritmos

El grafico muestra una relación entre la complejidad de cada algoritmo variando el parámetro k. Simplemente a modo de ejemplo, se eligió un tamaño de entrada determinado ($n = 16$).

Ahora bien, fuera del tamaño determinado, podemos rescatar que hay 3 algoritmos que no tienen un determinado k para el cual alguno es mejor que el otro. Es el caso del algoritmo de fuerza bruta, el de order and select y el QuickSelect. Lógicamente, el algoritmo de fuerza bruta no puede competir contra absolutamente nadie. Solo resta decir para el mismo que la complejidad de el k-selections se equipara con el de fuerza bruta cuando $k = n$.

Para valores pequeños de k, observamos que el algoritmo más óptimo es el HeapSelect. A medida que aumentamos el k pedido, éste algoritmo es superado por el k-heapsort. Partiendo de la complejidad de los dos algoritmos, esto comienza a ocurrir a partir de que $k * \log(n) = n * \log(k) -> k/\log(k) = n/\log(n)$. Luego, podemos ver que el QuickSelect es el que termina siendo el más óptimo para valores de k más altos. Tomando los órdenes de complejidad mostrados anteriormente, esto ocurre a partir de que $n = k * \log(n) -> k = n/\log(n)$. Para el ejemplo dado con $n = 16$, esto comienza a ocurrir para $k >= 6$.

Es de notar también que para valores altos de k, tanto el k-heapsort como el HeapSelect convergen al mismo orden que el Timsort utilizado. También vemos que el k-selections es más eficiente que el Timsort para elecciones k menores a 3 (igualando la complejidad calculada de ambos, se da cuando $n * \log(n) = k * n -> \log(n) = k$, lo cual da un $k \sim 277$ para el ejemplo).

2. Camino mínimo en Grafos

Los problemas de camino mínimo en grafos consisten en encontrar una secuencia de aristas desde un vértice de origen, al que llamaremos s a uno de llegada, al que llamaremos t, que sea la secuencia de menor costo total posible. El costo total de un camino será la suma de los costos de cada una de las aristas. Abajo se desarrollan 4 algoritmos posibles para lograr este objetivo.

Si bien todos los algoritmos calculan la distancia del camino mínimo, es posible reconstruir ese camino si en el proceso se guarda en cada vértice el padre desde el cual se llegó a él. De esta manera, al finalizar, queda construida la lista hacia atrás de padres desde el origen que conforma el camino mínimo.

2.1. Búsquedas No Informadas

Los primeros dos algoritmos que utilizaremos no conocen nada del problema modelado y por lo tanto no saben ni pueden estimar sobre lo que habrá más adelante en el grafo. Comienzan desde el vértice t y su frontera de conocimiento sobre el grafo se va expandiendo desde allí.

2.1.1. BFS

BFS es un recorrido en anchura de uso general para grafos no pesados (o de pesos iguales, lo cual es equivalente).

Se puede utilizar para medir caminos mínimos en este tipo de grafos y la solución devuelta es verdaderamente la óptima. En grafos pesados, sin embargo, no queda garantizado que la solución sea correcta, y en general no lo es: se obtendrá un camino de s a t, pero la distancia estará mal calculada o podrá no ser la mínima.

Este recorrido utiliza una cola para guardar todos los nodos adyascentes a cada vértice, y luego desencolar un vértice de esa cola para repetir el proceso con los aydascentes del nuevo. De esta forma, la cola comienza únicamente con el vértice s, se desencola, se encolan sus vecinos, y comienza el recorrido desde allí.

En el caso de su uso para búsqueda de camino mínimo, se etiqueta a cada vértice al ser encolado con un nivel: una distancia discreta a s, el primer vértice encolado, se lo etiqueta con nivel 0. Luego, cada vértice encolado es etiquetado con el nivel de su padre incrementado en 1. Al llegar el momento de encolar t, su nivel será la longitud del camino mínimo.

2.1.1.1. Optimalidad

Es importante notar que se recorre por niveles. Es decir, siempre se recorrerán primero los vértices de distancia 0 (s), luego se encolarán todos los vértices de nivel 1, luego se encolarán los de nivel 2 (y antes de llegar a los de nivel 2 se habrán recorrido todos los del nivel 1, gracias al orden FIFO) y así sucesivamente. Esto permite asegurar que nunca se dará el caso de encontrar un camino alternativo a un vértice que sea menor al antes calculado.

Esto no seguirá valiendo para grafos pesados, donde para asegurar aquello habrá necesidad de utilizar una cola de prioridad, y en eso se basa Dijkstra.

2.1.1.2. Complejidad

Teniendo en cuenta el algoritmo desarrollado, lo que se hace es recorrer los vertices una sola vez (se agregan a la cola si y solo si no fueron ya agregados antes) cortando cuando se llega a destino. Además se recorren las adyacencias de cada uno también una sola vez, por lo que se visitan las aristas una sola vez. En el peor

de los casos, el nodo destino es el último que se encuentra, y se habrán recorrido todos los nodos y aristas adyascentes, por lo que este algoritmo es $O(|V| + |E|)$.

2.1.2. Dijkstra

Dado un Grafo G y un vértice u del mismo, el algoritmo calcula la distancia de u a todos los otros vértices de G. Opcionalmente, para calcular el camino mínimo entre dos vértices u y v, se puede utilizar Dijkstra hasta que se llegue a calcular la distancia a v y se lo detiene ahí.

2.1.2.1. Optimalidad

Al igual que BFS para caminos mínimos, Dijkstra se basa en que en todo momento, si se desencola un elemento, se asegura que se llegó a él con el menor costo posible. Para esto se utiliza una cola de prioridad, ordenada por el costo total en llegar desde s hasta cada vértice por el camino desde el cual se lo agregó. Llamaremos a esta distancia $g(v)$.

De este modo nos aseguramos que al visitar un vértice (desencolarlo) se está llegando a él con el menor costo posible (si hubiese un costo menor, ya se lo habría visitado antes). Así como BFS aseguraba que primero se visitaran todos los elementos de nivel 1, luego los de nivel 2, etc, Dijkstra asegura que siempre se visitarán los elementos de menor distancia a s antes de los de mayor distancia.

Entonces la prioridad se calcula con $g(v)$, vértice a vértice, incrementalmente.

Si se encuentra un camino alternativo a un vértice con costo menor, se lo agrega a la cola otra vez, con su nueva prioridad, con lo cual la alternativa más corta será la primera en ser visitada. Esto puede verse como una "actualización de prioridad".

A su vez, nunca ocurrirá que encontremos un camino alternativo más corto a un vértice ya visitado, ya que si así fuera, el subcamino hacia su padre nuevo habría sido visitado antes que él y habría sido agregado con esa prioridad antes de ser visitado.

Estos puntos, teniendo en cuenta que **los pesos son positivos**, aseguran la optimalidad del algoritmo. Si hubiera pesos negativos, este algoritmo ya no sirve, porque bien podría ocurrir que se encuentrara un camino que supere al anterior en peso, y que al final tenga una arista que compensaba.

2.1.2.2. Complejidad

El algoritmo de Dijkstra se desarrolló utilizando una cola de prioridad implementada con un heap. Por lo que la extracción del siguiente nodo a procesar (menor distancia == raiz) es $O(1)$ y la inserción es $O(log|V|)$. Ahora bien, en el peor de los casos, cada vertice está conectado a |V| - 1 vértices, que equivale a la cantidad de aristas por vértice. Habiendo implementado un heap, si tengo que encontrar y actualizar el heap por cada arista, entonces $O(|E| * (1 + log|V|))$ -> $O(|E| * log(|V|))$. Ahora, esto es iterativo para cada uno de los vertices del grafo (en el peor de los casos), por lo que sería $O(|V| * |E| * log(|V|))$. Finalmente, puede ajustarse a $O(|E| * log(|V|))$.

2.2. Best First Search

Los dos algoritmos siguientes forman parte de una familia llamada *Best First Search*, que ordena el siguiente nodo a visitar con una función de evaluación f(n) que toma información inherente al problema modelado. Esta es una familia de algoritmos de *Búsqueda informada*.

Estos algoritmos, entonces, a diferencia de los anteriores, no utilizan solo los datos de la estructura del grafo, si no de lo que representan en el modelo. Estos datos en general permiten estimar la distancia entre los nodos según la distancia en términos del problema, con una función llamada *heurística*.

2.2.1. Heurísticas

Una heurística $h(u, v)$ es una función que *estima* la distancia entre los vértices u y v.

La forma más básica de Best First Search es la usualmente llamada greedy, que consiste en ordenar únicamente los nodos a visitar por su distancia estimada al nodo t. Esto es: $f(n) = h(n, t)$ De este modo, en lugar de recorrer por niveles de lejanía al origen, se recorre según la aparente cercanía al destino.

El diseño de h dependerá del problema modelado. Si, por ejemplo, el problema tiene que ver con el mapa de una ciudad o un laberinto, una heurística puede ser la distancia Manhattan. En un recorrido a gran escala, se podría usar, por ejemplo, el arco de circunferencia mínimo entre s y t.

Es importante notar que para que la estimación tenga sentido, los nodos deben tener información de estado (en los ejemplos, la posición física) y los pesos de las aristas deben también responder al modelo (por ejemplo, con la distancia real entre dos nodos conectados).

Del mismo modo, cabe aclarar que en problems de recorrido físico, h no debe estimar la distancia entre las posiciones de cada nodo, sino la distancia recorrible entre ambos **en el grafo**. Por ejemplo, en una manzana de Manhattan, si bien la distancia real entre dos esquinas es la diagonal que cruza por el centro de la manzana (distancia euclídea), la distancia que debemos estimar es la que efectivamente costará llegar de una esquina a la otra, y esta tendrá que ser dando la vuelta a la manzana.

2.2.1.1. Optimalidad

Esta búsqueda, solo con heurísticas, no asegura optimalidad, aunque mejora con la precisión de la estimación de h sobre la distancia real del camino.

Si la distancia estimada fuera *exactamente igual* a la distancia real en el grafo, entonces recorreríamos primero un camino mínimo y obtendríamos una resolución óptima.

2.2.2. A*

A* es una mejora de Dijkstra, utilizando heurísticas. Así como Dijsktra ordena según la distancia total hasta un vértice $g(n)$ y la búsqueda con heurísticas minimiza $h(v, t)$, A* ordena según la suma de ambas. Esta función de evaluación es entonces:

$$f(v) = d(s, v) + h(v, t) \tag{1}$$

De este modo, el orden tiene en cuenta tanto lo ocurrido como la estimación de lo que va a ocurrir por ese camino, lo cual permite una decisión más informada que Dijkstra.

A* no es óptimo según cualquier heurística, y por eso hay dos propiedades importantes a estudiar.

2.2.2.1. Admisibilidad

Una heurística es admisible cuando nunca sobreestima una distancia.

$$h(v,t) \leq d(v,t) \tag{2}$$

Si en la búsqueda permitiéramos expandir un nodo múltiples veces (lo que se suele llamar búsqueda en árboles) entonces se puede probar que una heurística hace que A* sea óptimo. Puede demostrarse que siempre se elegirá un vértice de camino óptimo antes que un estado final t por un camino subóptimo.

Para el diseño de heurísticas admisibles se suele tomar la estrategia de *problema relajado*, que consiste en simplificar el problema, para asegurarse de que la estimación subestime a la distancia.

En un laberinto, por ejemplo, utilizar la distancia Manhattan entre dos puntos consiste en la suposición de que no hay paredes. De esta manera, la distancia estimada siempre será menor a la real, donde habrá obstáculos, y por lo tanto, será admisible.

Como propiedad adicional, si una heurística es admisible se cumple que $h(u,u) = 0$, lo cual hace que la función de evaluación en t tenga heurística 0.

2.2.2.2. Consistencia

Que una heurística sea admisible, sin embargo, no es suficiente para poder ejecutar el algoritmo de la misma manera que Dijkstra, visitando una sola vez a cada nodo y sin necesidad de tener que actualizarlo. Para poder hacer esto, la heurística debe ser **consistente**.

Una heurística es consistente cuando se cumple que:

$$h(u,w) \leq h(v,w) + d(u,v) \tag{3}$$

donde u, v y w son tres nodos cualesquiera. Se suele utilizar esta propiedad con $w = t$.

Esta propiedad es intuitivamente similar a la desigualdad triangular:

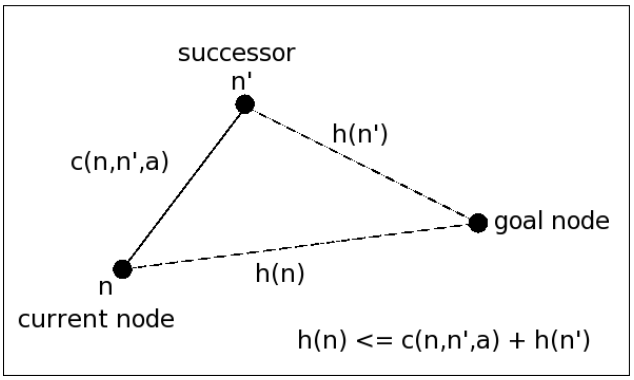


Figura 11: Comparación de la desigualdad triangular con la consistencia.

Las heurísticas consistentes se llaman también monótonas porque se puede probar que en un camino cualquiera la función de evaluación f(v) = g(v) + h(v,t) es no decreciente a lo largo del camino. Esta propiedad permite

ordenar según f(v) al igual que en Dijkstra se ordena según $g(v)$, sabiendo que al ser visitado un vértice, no habrá un camino alternativo más corto (o en este caso con menor f) que el que se está expandiendo.

Además, se puede probar fácilmente por inducción que una heurística consistente también es admisible.

Obs: en el caso de $h(v) = 0 \quad \forall v$, se cae en el caso de Dijkstra, que es consistente.

2.2.2.3. Optimalidad

Para que una ejecución de A* sea óptima (que devuelva el mejor camino) y que no se repitan vértices, la heurística utilizada debe ser consistente, y por lo tanto, también admisible.

Si bien es normal que las admisibles sean también consistentes, no ocurre siempre de este modo y es algo que hay que demostrar según el caso.

2.2.2.4. Eficiencia

Cuanto más se acerque una heurística consistente a la verdadera distancia, las elecciones que se tomen primero serán más probablemente las del camino óptimo y por lo tanto se llegará antes al destino t. De este modo, Dijkstra constituye la peor heurística consistente para A*.

Por este motivo, en el problema del laberinto, si bien la distancia euclídea es admisible (supone que no hay paredes y que uno se puede mover en cualquier dirección), subestimará aún más a la distancia y por lo tanto es probablemente menos eficiente que la distancia Manhattan.

De hecho, como $h_{euclidea} \leq h_{manhattan} \quad \forall (u, v)$, se dice que la heurística con distancia Manhattan **domina** a la distancia euclídea y por lo tanto es mejor en cualquier punto.

Finalmente, si se tienen dos buenas heurísticas y ninguna domina a la otra, se puede calcular el máximo entre ambas, y esto dará una mejor heurística (y puede probarse que se mantiene la consistencia).

2.3. Ejemplos

2.3.1. Camino con Heurísticas contra A*

Previamente se mencionó que mientras la búsqueda con Heurísticas plana se fija únicamente en lo estimado de lo que ocurrirá a futuro, mientras que A* tiene en cuenta además lo que ocurrió hasta el momento. Esta diferencia puede verse clara en el grafo de la figura 12.

Para la búsqueda con heurísticas se ve muy bien que ir por el camino de la derecha se acerca más rápido hacia el destinto t. Sin embargo, el desvío producido a partir del nodo 3 no empeora la distancia lo suficiente como para explorar a partir del nodo 14. Como este algoritmo no tiene en cuenta cuánto ya recorrió, entonces nunca explorará el camino de la izquierda, ya que estima la distancia estimada desde allí será de 6, mientras que a la derecha nunca supera 5.

Por este motivo, el programa ejecuta y muestra:

```
Camino con búsqueda por heurísticas:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]
Camino con A*:
[0, 14, 15, 16, 17, 18, 19, 13]
----.----
```

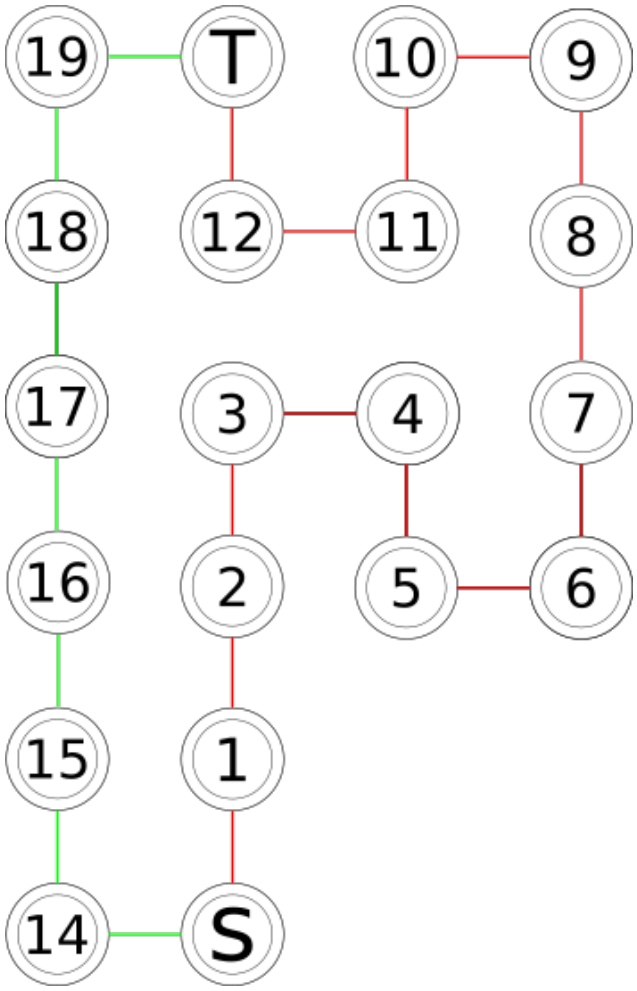


Figura 12: Recorridos en A* y en Heurísticas

2.3.2. Heurísticas admisibles y no admisibles

También fue mencionado anteriormente que las heurísticas que sobreestiman la distancia entre dos puntos no son admisibles y calculan caminos subóptimos en A*. En la figura 13 se ve un grafo en forma de triángulo, y se ven las heurísticas y funciones de evaluación para cada vértice.

En una de la arista entre el nodo 1 y el t (nodo 3) la arista es diagonal, y la heurística Manhattan (la línea punteada) sobreestimará lo faltante. Por este motivo, A* con distancia manhattan elegirá el camino de arriba, ya que estima que para el vértice 2 faltan 9, mientras que para el vértice 1 faltan 21 y por lo tanto las funciones de evaluación entonces son 20 para el camino de arriba y 22 por el camino de abajo.

Para la distancia euclídea, sin embargo, la distancia estimada para el vértice 1 es efectivamente 15 y elegirá el camino de abajo. La distancia total sería entonces 20 para la heurística Manhattan y 16 para la heurística Euclídea, que no sobreestima la verdadera distancia.

De este modo se observa que la distancia Manhattan puede dominar a la Euclidea y ser consistente para los problemas de grillas, pero que en un problema libre como este, no es ni siquiera admisible.

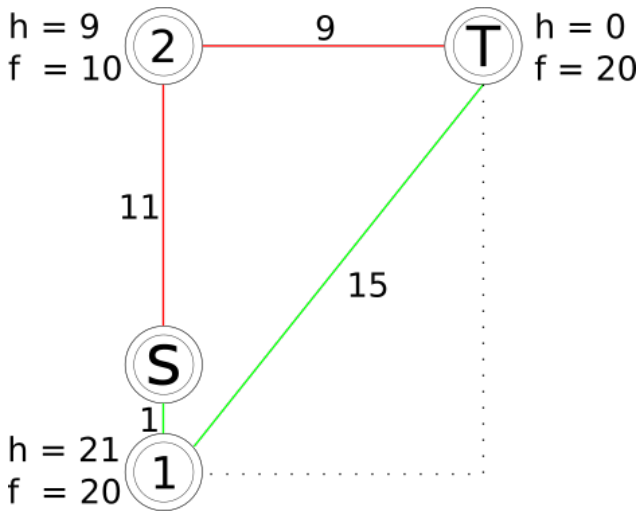


Figura 13: A* con Heurística Manhattan y Euclídea para un mismo problema.

Como lo predicho, al ejecutar el programa la salida es la siguiente:

```
Camino de A* con Heurística Manhattan
[0, 2, 3]
Camino de A* con Heurística Euclídea
[0, 1, 3]
```