Teoría de Algoritmos - Trabajo Práctico 1

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

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 4to 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. Sort 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 lineal 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).

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

TODO

$O(n \log n)$

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 Sort 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

1.4.3. Peor caso

1.5. HeapSelect

Esta es una función de `Python` que se encarga de algo parecido:

1.5.1. Nota sobre `heapq.nsmallest`

Este algoritmo de `Python` que se encuentra en el repositorio 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. Ordena el heap de k elementos, lo cual es $O(k \log k)$ y devuelve esa lista.

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

Como explica la documentación, 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. Solución Diseñada

No nos conviene usar el `heapq.nsmallest`, ya que el 3er paso ordena los k más pequeños, siendo que nosotros solo necesitamos obtener el máximo de estos últimos. Nos basaremos en esta documentación para implementarlo, pero variando el paso 3. En aquel paso solo observaremos el máximo, que justamente es la cabeza del heap. Esto es $O(1)$ y es lo que se devuelve.

El orden queda entonces: $O(k + n \log k + 1) = O(n \log k)$. Es básicamente la función de `Python`, pero sin el orden del final, lo cual reduce sustancialmente el tiempo.

1.5.3. Mejor caso

1.5.4. Peor caso

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

1.9. Elección de algoritmo óptimo para cada 'k' según 'n'

2. Camino mínimo en Grafos

Para todos los casos llamaremos s al nodo de salida u origen y t al nodo de llegada u objetivo.

2.1. 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 comineza 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. 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 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.

2.1.2. Reconstrucción

Tanto para este algoritmo como para otros, para reconstruir el camino basta guardar los padres de cada nodo encolado para luego armar la lista hacia atrás desde el destino.

2.2. Heurísticas

La búsqueda con Heurísticas utiliza un criterio distinto para recorrer el grafo. En lugar de una cola común utiliza una cola de prioridad, ordenada según una función $h(u, v)$ llamada Heurística, que *estima* la distancia entre los vértices u y v. De este modo, en lugar de recorrer por niveles, se recorre según lo estimado, recorriendo siempre primero lo que *aparenta* estar más cercano del destino.

La función heurística requiere un diseño que no es inherente al grafo como estructura abstracta, sino que responde al problema que se está modelando. 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 podríausar, por ejemplo, el arco de circunferencia mínimo entre s y t.

2.2.1. Optimalidad

La búsqueda con heurísticas no asegura optimalidad y es tan buena como lo sea la estimación de h sobre la distancia real del camino. Por ejemplo, en una manzana de Manhattan, si bien la distancia real entre dos esquinas es la diagonal que cruza 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. Entonces es importante tener en cuenta que **la distancia estimada es la del camino en el grafo**, no la *distancia física* si la hubiera.

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

2.3. Dijkstra

Dado un Grafo G y un vértice v del mismo, el algoritmo calcula la distancia de v 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 u y se lo detiene ahí.

2.3.1. Optimalidad

Al igual que BFS para caminos mínimos, 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 hasta cada vértice. 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: $f(v) = d(s, v)$, donde $d(s, v)$ se calcula 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 optimalidad. Si hubiera pesos negativos, este algoritmo ya no sirve, porque bien podría ocurrir que se encuentre un camino que superaba al anterior en peso, y que al final tenía una arista que compensaba.

2.4. A*

A* es una mejora de Dijkstra, utilizando heurísticas. Así como Dijsktra minimiza la distancia total hasta un vértice, A* minimiza la distancia de s hasta ese vértice sumada a la distancia estimada de ese vértice a t (la heurística) y ese es su criterio para el orden en la cola de prioridad. Numéricamente la prioridad $f(v) = d(s, v) + h(v, t)$.

De este modo, lo minimizado tiene en cuenta lo ocurrido y la estimación de lo que va a ocurrir, lo cual permite una decisión más informada que Dijkstra.

2.4.1. Optimalidad

Para que se mantenga la optimalidad que brindaba Dijkstra, se debe cumplir que la heurística sea **admisible**, es decir que nunca sobreestime la distancia faltante. Numéricamente $h(v, t) <= d(v, t)$.

Por este motivo se suele tomar la estrategia de *problema relajado* para el diseño de heurísticas. Esta estrategia 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.

`TODO: explicar por qué una heurística admisible permite optimalidad.`

2.4.2. Eficiencia

Por otro lado, cuanto más se acerque una heurística admisible 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 destinto t.

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), pero subestimará aún más a la distancia y por lo tanto es probablemente menos eficiente que la distancia Manhattan.