Pancake Sorting

Hecho por: Juan David Guevara Arévalo

Algoritmo de solución

Para solucionar el problema, se optó por utilizar una aproximación con grafos. Inicialmente, era evidente la necesidad de descartar el uso de algoritmos como Dijkstra, Bellman Ford o alguno de los estudiados en clase, pues el espacio de búsqueda de estos algoritmos es $n!$, ya que se deben explorar todas las posibilidades. Sin embargo, se tomó como inspiración el algoritmo A* (aStar), el cual es un algoritmo aproximado para la búsqueda del camino más corto, el cual utiliza una combinación entre el algoritmo de Dijkstra y BFS para realizar una búsqueda informada.

A* es un algoritmo que comúnmente consta de una función de coste, una función de coste aproximado (heurística) y una función de evaluación. Normalmente, la función de coste evalúa el peso real del camino revisado. En un laberinto, podría ser la cantidad de celdas visitadas en un camino. Por otro lado, la función de coste aproximado tiene más libertad en cuanto a lo que evalúa; sin embargo, debe respetar las proporciones de la función de coste real, de esta manera, se puede dar un estimado del peso real restante. En el ejemplo del laberinto, la función de coste aproximado podría ser la distancia de Manhattan que hay desde cualquier punto hasta el punto de salida. Finalmente, la función de evaluación determina que paso tomar según los resultados de las funciones de coste. En el caso del laberinto, podría ser `ditanciaReal + distanciaManhattan`, de esta manera se da prioridad a visitar los nodos más cercanos (o lo que cercano signifique para la heurística definida) al punto objetivo y así se minimizan la cantidad de estados visitados, los cuales nuevamente dependerán de la heurística seleccionada.

Diseño de la heurística

Una vez se escogió A* para resolver el problema, fue necesario diseñar diversas heurísticas para que el algoritmo de A* fuera los suficientemente eficiente.

La primera aproximación fue la más sencilla posible: medir la cantidad de pancakes que están en la posición correcta, sin embargo, esta estrategia fue descartada rápidamente debido a que no apunta en ninguna dirección, pues, el stack `1 2 3 4 5` sería "peor" que el stack `5 3 4 2 1`, aun cuando es evidente que hace falta un solo paso para ordenar el primero.

Tras varias pruebas, la primera aproximación útil fue contar la cantidad máxima de elementos ordenados en ambas direcciones. Así, para los anteriores ejemplos, el primer stack tendría un puntaje igual a 5 y el segundo igual a 2. Finalmente, tras ligeros cambios y aproximaciones siguiendo la misma idea de "elementos ordenados", se obtuvo lo que en adelante se llamará **Paired Neighbors**.

Heurística "Paired Neighbors"

Esta heurística conseguida es sencilla, pero eficiente. A continuación, habrá una explicación del porqué la heurística de los vecinos emparejados es eficiente no solo en términos de aproximarse rápidamente al objetivo sino del costo que tiene calcular su valor.

Este cálculo consiste en tomar una pareja de pancakes consecutivos en el stack a revisar, y ver si en el stack objetivo ambos pancakes de dicha pareja son vecinos (de ahí el nombre elegido para la heurística), por lo que, entre más vecinos emparejados haya, mejor será el stack de pancakes. Lo anterior permite que dos estados tengan el mismo puntaje si uno es el reverso del otro. *i.e.* `1 3 4 2` = `2 4 3 1`.

Tras implementar esta heurística y realizar un análisis de los resultados, el algoritmo A* no funcionaba nada bien, pues parecía no avanzar y tenía en cuenta cerca de los $n!$ posibles estados antes de encontrar la solución.

Para solucionar dicho problema, se analizó la función de optimización, la cual une la heurística con el peso real, el cual se tomaba como la cantidad de flips utilizados en el camino a revisar. Sin embargo, sin importar cuanto se avanzara, el resultado de dicha función era siempre el mismo, razón por la cual, luego de estudiar la heurística con la naturaleza de los flips, se concluyó que tras un flip, la cantidad de vecinos emparejados

podría verse afectada en tres aspectos: aumentar en uno, disminuir en uno o mantenerse intacta. (Más adelante hay una demostración disponible).

Lo anterior resultó en que se eliminara la función de peso real, ya que "los vecinos emparejados" cómo heurística es lo suficientemente buena para que funcione por si sola, sin embargo, para diferenciar de algún modo el estado en orden inverso del estado en orden normal, se considera el "plato" y el primer pancake como una pareja más, resultando en n parejas para un stack de n pancakes. Además, otro resultado es que una vez se tienen los vecinos emparejados para el stack inicial (en una complejidad $O(nlogn)$), es posible calcular la cantidad de vecinos emparejados para cualquier flip posible en $O(logn)$, lo que hace de la heurística algo muy eficiente. (Se utiliza búsqueda binaria para determinar si dos elementos son vecinos en un arreglo ordenado).

Demostración "Paired Neighbors"

Sea S un stack con n pancakes, donde P_i es el pancake en la posición i de la pila S.

Entonces: $S \equiv BP_0, P_1, P_2, \ldots, P_{i-1}, P_i, P_{i+1}, P_{n-2}, \ldots, P_{n-1}, P_n$

Donde B es la base de la pila y no se puede mover.

Además, sea $f(S, i)$ el flip de S en el índice i, $v(S)$ la función que calcula el valor de la heurística para la pila S y $a(S, i, j)$ la función que determina si el pancake i y el pancake j son vecinos en S.

Entonces: $f(S, i) \equiv B, P_0, P_1, P_2, \ldots, P_{i-1}, P_n, P_{n-1}, P_{n-2}, \ldots, P_{i+1}, P_i$

Para el cálculo de los vecinos emparejados suponga S_o una pila con 4 pancakes correctamente ordenada, la cual será el objetivo, y S la pila inicial a ordenar.

$\rightarrow S_o = B, 4, 3, 2, 1 \wedge S = B, 3, 1, 2, 4$

Pasos:

¿La pareja (B,3) son vecinos en S_o? No

¿La pareja (3,1) son vecinos en S_o? No

¿La pareja (1,2) son vecinos en S_o? Sí, pues en S_o tenemos $(2, 1)$

¿La pareja (2,4) son vecinos en S_o? No

Por lo tanto: $v(S) = 1$

Ahora, se aplica $f(S_n, i)$ con S_n una pila ordenada con $v(S_n) = n$.

$$S_n = BP_0, P_1, P_2, \ldots, P_{i-1}, P_i, P_{i+1}, P_{n-2}, \ldots, P_{n-1}, P_n$$

$$\langle S_{ni} = f(S_n, i) \rangle$$

$$S_{ni} = B, P_0, P_1, P_2, \ldots, P_{i-1}, P_n, P_{n-1}, P_{n-2}, \ldots, P_{i+1}, P_i$$

Es fácil observar que la función $a(S, i, j)$ luego de $f(S, i)$ se mantendrá igual desde B hasta $i - 1$

$$a(S_n, B, 0) \equiv a(S_{ni}, B, 0) \wedge a(S_n, 0, 1) \equiv a(S_{ni}, 0, 1) \wedge \ldots \wedge a(S_n, i-2, i-1) \equiv a(S_{ni}, i-2, i-1)$$

Sin embargo, si recorremos S_{ni} en sentido opuesto encontramos las siguientes parejas:

$(P_i, P_{i+1}), (P_{i+1}, P_{i+2}), \ldots (P_{n-1}, P_n), (P_n, P_{i-1})$

Ahora, podemos observar que de nuevo se repiten parejas. En concreto, **todas las parejas son iguales, excepto porque desaparece** (P_{i-1}, P_i) **y aparece** (P_{i-1}, P_n), lo que nos permite reutilizar el valor de $v(S)$ antes de realizar cualquier $f(S, i)$ para calcular el siguiente tan solo con revisar las 2 parejas que cambian. Por lo que, en particular, calcular el mejor flip para una pila S cuyo $v(S)$ es conocido toma $O(nlogn)$

Resultados

Tras ejecutar el algoritmo en un amplio set de datos de prueba (*más de 100000 pilas distintas con tamaños desde 20 hasta 1000 pancakes*), se obtiene que la cantidad de flips promedio que se necesita para ordenar cualquier pila de n pancakes con la variante de A* y la heurística de los vecinos emparejados desarrollados yace entre n y $1.2n$ flips.

Complejidades

La complejidad temporal del algoritmo A* depende enteramente de la heurística seleccionada, y si la heurística es mala, el algoritmo visitará todos los posibles estados, en este caso $n!$. Sin embargo, `vecinos emparejados` permite que el número de pilas revisadas sea nš, pues la mayoría de las veces para cada pila elegida revisará todas las opciones de flip, elegirá una, y para el flip seleccionado repetirá el proceso. Lo anterior fue comprobado directamente en el algoritmo (*con cientos de pilas de diferentes tamaños*), y el promedio obtenido fue $n^{2.0857}$.

Complejidad temporal

Lo anterior nos permite decir que revisamos nš estados, para cada estado revisamos n posibilidades de flip, y calcular el resultado de cada flip toma n pasos. Por lo que la complejidad temporal del algoritmo es n (Ver Anexo 1).

Complejidad espacial

Cada estado es un arreglo con n elementos, por lo que la complejidad espacial del algoritmo es nş

Escenarios

1. Switch en una pila

Utilizando el algoritmo actual, habría que incluir los nuevos posibles estados, pues para cada camino podemos considerar $\frac{n(n+1)}{2}$ posibles intercambios, sin embargo, esto no resultaría muy viable. Para tener una mejor aproximación, podemos considerar los intercambios una vez el valor de la heurística esté lo suficientemente cerca de n (cota), momento en el que los intercambios podrían ser más útiles. Un buen valor para empezar a considerarlos podría ser $0.8n$.

2. La peor pila

Dado que la heurística nos ayuda a determinar que tan buena es una pila, podemos aplicarla para minimizar su valor, llegando a un resultado que seguramente tendrá intercalados los pancakes más grandes con los más pequeños, pero intercambiando el pancake del medio con el más grande. `i.e. 5 1 8 2 7 3 6 4 9` el cual tiene 0 vecinos emparejados.

Consideraciones

1. Las pilas utilizadas para probar el algoritmo fueron generadas sintéticamente mediante el uso de un programa escrito en el lenguaje Rust, utilizando el módulo `rand`.
2. Los valores promedio fueron calculados sobre conjuntos de pilas con tamaños que variaban entre 10 y 200 pancakes.
3. Para estimar la cantidad de estados visitados se utilizó un conjunto de cien mil pilas con tamaños variados, y el valor tomado para calcular el resultado fue el tamaño de la tabla de hash una vez se encontró la solución a cada caso, utilizando el logaritmo base n de la cantidad de elementos utilizados.
4. Se excluyeron de los cálculos los casos que ya se encontraban con demasiados vecinos emparejados en un inicio.

Anexos

El gráfico se generó utilizando `python` con la librería `matplotlib`. Los tiempos se tomaron utilizando el programa realizado en `Java` (El envío unicamente contiene el algoritmo principal) con la función `System.currentTimeMillis()`.

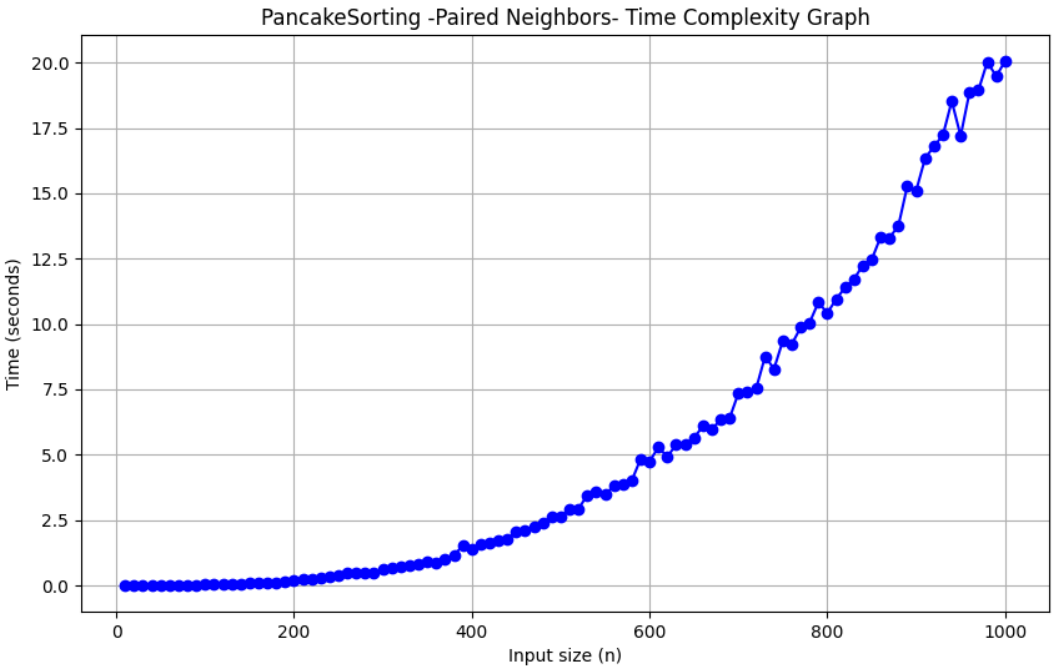

Figure 1: Tiempo en función de n

Comentarios adicionales

La implementación del algoritmo fue realizada en Java. El programa genera una visualización paso a paso para el primer caso de prueba. Las visualizaciones son generadas en el formato `.dot`. Para visualizarlo se puede exportar a jpg utilizando GraphViz o tambien se puede utilizar un Playground en linea.