Entrega 2 Proyecto Dalgo

Luis Alejandro Rubiano 202013482
Isaac Bermudez 202014146
Sebastián Gaona 202012158

I. Explicación algoritmo

El algoritmo consiste en 3 partes principales:

1. Ordenamiento de las páginas del diccionario.
2. Construcción del grafo dirigido.
3. DFS para ordenamiento topológico con detección de ciclos.

I-A. Preliminares

1. Entrada y salida de datos.

   Los datos se leen por salida y entrada standard de Python, ejemplo de uso del archivo sería hacer cd en terminal a la carpeta en donde se encuentra ProblemaP2.py y ejecutar:

   python3.10 ProblemaP2.py $<$ P0.in $>$ archivo2.out

   Nota: Este algoritmo fue implementado y testeado en Python 3.10, se recomienda usar esta versión para poder reproducir los resultados con precisión.

2. Explicación de la solución
   El problema que nos ponen es equivalente a encontrar si existe un orden lexicográfico dado un conjunto de palabras.
   El enunciado equivalente, sería una lista de palabras, determine si existe un orden lexicográfico f.
   Sea f un orden parcial, sabemos que un orden parcial puede ser representado por un DAG. Ahora sabemos que la tarea se reduce si podemos contruir un DAG a partir de las palabras que nos dan.
3. Implementación de estructura de grafo.

   Para este problema se creó una implementación de estructura de datos de Python para representar a un grafo dirigido.

   Esta es una Clase de Python con los siguientes atributos: Grafo.V (set() de Python - estructura de datos hasheada - que representa todos los vertices en el grafo - caracteres unicode), Grafo.A (dict() de Python que contiene parejas llave:valor de tipo vertice:[lista de vertices] en donde si $v\ in\ Grafo.A[u]$ entonces existe un eje dirigido (u, v) en el grafo), Grafo.marcados_temporal (set() de Python subconjunto de Grafo.V - más adelante se explicará mejor), Grafo.marcados_permantente (set() de Python subconjunto de Grafo.V - de igual forma más adelante se explicará mejor) y Grafo.contiene_ciclo (bool que es True si se detecta un ciclo en el grafo, False de lo contrario). De igual manera, la clase Grafo contiene los siguientes métodos: Grafo.anadir_vertice() y Grafo.anadir_eje() (hacen lo que su nombre indica), Grafo.__str__() y Grafo.__repr__() (para propositos de debugging), Grafo.dfs_topological_sort() y Grafo.dfs_topological_sort_util() (para procesos algoritmicos que se explicarán más adelante).

```python
class Grafo:

    def __init__(self):
        self.V = set()
        # Diccionario de adyacencia nodo_de_origen:[nodos_destino]
        self.A = dict()
        self.marcados_temporal = set()
        self.marcados_permantente = set()
        self.contiene_ciclo = False

    def anadir_vertice(self, v):
        self.V.add(v)
        if v not in self.A:
            self.A[v] = []

    def anadir_eje(self, v1, v2):
        if v2 not in self.A[v1]:
            self.A[v1].append(v2)

    def __str__(self):
        return f"V: {self.V}, A: {self.A}"

    def __repr__(self):
        return f"V: {self.V}, A: {self.A}"

    def dfs_topological_sort_util(self, i, stack):
        if i in self.marcados_permantente:
            return
        if i in self.marcados_temporal:
            self.contiene_ciclo = True
            return

        self.marcados_temporal.add(i)

        for j in self.A[i]:
            self.dfs_topological_sort_util(j, stack)

        self.marcados_temporal.remove(i)
        self.marcados_permantente.add(i)
        stack.insert(0, i)

    def dfs_topological_sort(self):
        stack = []
        for i in self.V:
            if self.contiene_ciclo:
                return stack
            elif i not in self.marcados_permantente:
                self.dfs_topological_sort_util(i, stack)

        return stack
```

Implementación de la estructura de datos Grafo

I-B. Ordenamiento de las páginas del diccionario

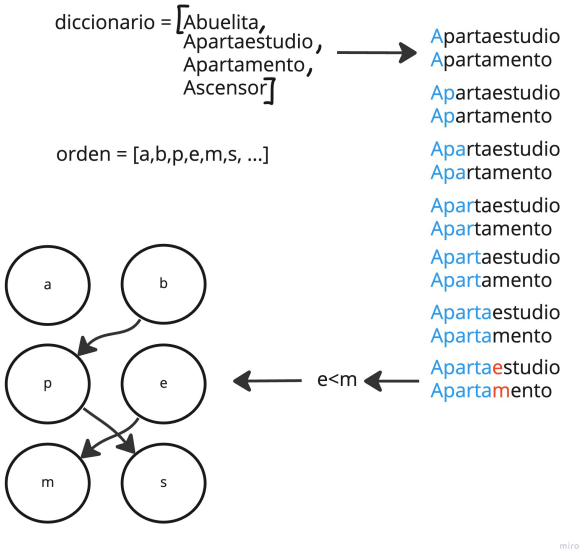
Para cada uno de los casos de prueba, se crea una lista de listas vacías, con el mismo número de listas vacías que el número de páginas en el 'diccionario'. Posteriormente, las pálabras de cada página se agregan a su posición correspondiente, por ejemplo, las palabras de página 2 se

añaden a la segunda lista de la lista de listas.

Posteriormente se realiza un proceso de 'aplanamiento' de los datos para llegar a una sola lista (no matriz) con las palabras ordenadas en todo el 'diccionario'. Ej: la lista $[['ab','ac'],['ad','ae']]$ se convertiría en $['ab','ac','ad','ae']$

I-C. Construcción del grafo dirigido

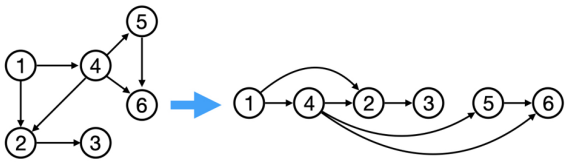
La construcción del grafo se realiza recorriendo el 'diccionario' de palabras en orden. Se define la variable $palabra\_anterior$ y a esta se le inicializa con el valor de la primera palabra en el 'diccionario'. Posteriormente se itera entre 1 y el número de palabras - 1. En cada iteración se le asigna a $palabra\_actual \leftarrow diccionario[i]$, luego se hace una iteración dentro de la previa entre los caractereres $palabra\_anterior$ y $palabra\_actual$ de manera paralela, desde 0, hasta $min(len(palabra\_anterior), len(palabra\_actual)) - 1$. En cada paso de este ciclo dentro del otro, se revisa si el caracter j-ésimo de $palabra\_anterior$ es igual al caracter j-ésimo de $palabra\_actual$, en caso de no ser iguales, se añade $(palabra\_anterior[j], palabra\_actual[j])$ como vértice al grafo, bajo la premisa de que $palabra\_anterior[j]$ va antes en el orden de las letras que $palabra\_actual[j]$. Finalmente, fuera del ciclo interno pero dentro del ciclo externo se reasigna $palabra\_anterior \leftarrow palabra\_actual$. A continuación se adjunta un dibujo para explicar el proceso anterior.



Ayuda visual para el algoritmo de construcción del grafo.

I-D. DFS para ordenamiento topológico con detección de ciclos.

Para concluír, una vez con el grafo ya construido se necesita una forma de ordenar las letras a partir de ese grafo. Una forma de hacer esto, que logra el objetivo de 'si la letra a va antes que b, entonces en el orden final la letra a debe ir antes que la b' es mediante un orden topológico. Particularmente en nuestro caso implementamos una modificación del algoritmo de DFS para orden topológico, en donde este orden es el mismo de las letras en el diccionario. La modificación al algoritmo es dividir los nodos marcados en marcados temporales y marcados permanentes para poder detectar ciclos en el grafo sin tener que recurrir a un algoritmo de detección de ciclos extra. En caso de que exista un ciclo en el grafo, se marca el atributo de $Grafo.contiene\_ciclo$ como $True$, para parar la ejecución; en este caso el algoritmo retorna error en el sentido de que el diccionario no se puede ordenar porque es inconsistente con el orden.



Ejemplo de Orden Topologico, tomado de LeetCode

II. Análisis temporal

La lectura de los datos se logra en tiempo:
$O(casos \cdot max(paginas))$

Y la complejidad espacial es $O(casos \cdot max(paginas) \cdot max(palabra))$ la creación de la matriz de páginas.

Construir el grafo de logra en tiempo
$O(casos \cdot max(paginas) \cdot max(len(palabra))$

porque se debe iterar sobre los caracteres de cada una de las palabras.

La complejidad espacial es de $O(casos \cdot max(letras)^2)$ donde letras es el número de letras. Esto viene de la construcción del $set()$ de vertices, y la lista de adyacencias de ejes.

El topological sort corre en tiempo
$O(casos \cdot max(letras) + max(letras)^2)$

(en el peor caso -grafo completo). La complejidad espacial de este paso es
$O(casos \cdot max(letras))$

por la construcción de la lista con el orden que se retorna.
Por lo tanto la complejidad temporal del algoritmo es
$O(casos \cdot max(paginas) \cdot max(len(palabra))$

y espacial es de
$O(casos \cdot max(paginas) \cdot max(palabra))$

III. Evaluación de escenarios

Escenario 1: Suponga que solo sabe el orden para un subconjunto de páginas

Partiendo de que el algoritmo diseñado consiste en comparar tuplas de caracteres con el fin de establecer su precedencia, se ha establecido como entrada una lista de palabras ordenadas por su página. El comportamiento descrito no debería cambiar en un contexto donde se desconoce el orden de un subconjunto de páginas. Estas simplemente no se tomarían en cuenta, ya que no aportan alguna información útil. Asímismo, para aquellos caracteres que únicamente se encuentren en el subconjunto de páginas que no se conoce el orden, el establecimiento de su precedencia es arbitrario.

Escenario 2: Suponga que existe la posibilidad que en el alfabeto de pandora se permitan los dígrafos como letras. Las famosas “ll” y “ch” que antes pertenecían a nuestro abecedario son ejemplos de digrafos

Para estos casos se ha establecido la posibilidad de reemplazar cada digrafo existente por algún caracter Unicode que no se encuentre en el alfabeto español, por ejemplo un ’/’ (Unicode 47) o una ’@’ (Unicode 64), con el fin de poder ejecutar el algoritmo y establecer la precedencia de este tipo de ’letra’ conformada por dos caracteres como si fueran uno. Los strings de Python soportan caracteres Unicode entre 0 y 1,114,111, entonces quitando los primeros 32 (reservados como los saltos de línea u espacios), quedan 1,114,079 posibles caracteres para asignar a alguna ’letra’ del abecedario, que finalmente es asociada a un vértice del gráfo.

Ej: En la palabra **Lleras**, en donde **Ll** es una ’letra’, se podria asociar **Ll** con @, y esta palabra quedaría **@eras**

El reemplazamiento anterior se podría realizar en tiempo $O(dnm)$ para cada string, en donde n es la longitud de la palabra, m la longitud del digrafo más largo y d el número de digrafos, usando algoritmos como Rabin–Karp (Rabin-Karp Algorithm, s. f.). Posteriormente este digrafo se podría reemplazar por un caracter unicode que no esté en uso en el alfabeto.

IV. Resultados y conclusiones

Se logró establecer un algoritmo para calcular precedencias de letras (alfabeto) a partir de la construcción de un grafo de implicación que representa ordenes parciales calculados mediante la comparación de caracteres de palabras ordenadas, seguido del uso una modificación de DFS para proveer un ordenamiento topológico con detección de ciclos a dicho grafo.

V. Referencias

Rabin-Karp Algorithm. (s.f.). https://www.programiz.com/dsa/rabin-karp-algorithm