Escuela Colombiana de Ingeniería Julio Garavito

Algoritmos y Estructuras de Datos 2024-1

Laboratorio MONTONES BINARIOS

Andersson David Sánchez Méndez

Cristian Santiago Pedraza Rodríguez

David Eduardo Salamanca Aguilar

17 de mayo de 2024

DOCUMENTO TÉCNICO

Requisitos

Especificación 1

Implementar el montón binario minimal (Completando la función min_heapify).

Especificación 2

Implementar las funciones de HeapSort para ambas versiones de montón binario (Maximal y minimal)

Especificación 3

Implementar el algoritmo de Dijkstra en donde en lugar de la función extrac_min se utilice la cola de prioridad para determinar aquel vértice con menor distancia.

Entrada general

Dado el prototipo implementado, en el mismo código se genera, por ejemplo, el nombre y edad de varias personas de forma aleatorio para que en la salida genere lo que está en el siguiente apartado. Y para la última especificación de entrada, un conjunto de relaciones que se va a hacer teniendo en cuenta grafos.

Salida general

Lo que quiero saber es atender a esas personas en orden de llegada, o por orden de edad. De esto depende si es minimal o maximal ese montón binario. También se harán casos de prueba para las 3 especificaciones descritas anteriormente. Teniendo en cuenta grafos y montones binarios.

Diseño

Estrategia 1

- **Identificación de los hijos y el nodo actual:**

  Se calculan los índices de los hijos izquierdo y derecho del nodo actual.

  left_index, right_index, smallest_index = self.left(index), self.right(index), index

- **Comparación con el hijo izquierdo:**

  Se comprueba si el índice del hijo izquierdo está dentro de los límites del montículo y si el valor del hijo izquierdo es menor que el valor del nodo actual.

  if left_index < len(self) and self.data[smallest_index] > self.data[left_index]: smallest_index = left_index

- **Comparación con el hijo derecho:**

  Similar al paso anterior, se comprueba si el índice del hijo derecho está dentro de los límites del montículo y si el valor del hijo derecho es menor que el valor actual más pequeño (ya sea el nodo actual o el hijo izquierdo).

  if right_index < len(self) and self.data[smallest_index] > self.data[right_index]: smallest_index = right_index

- **Intercambio si es necesario:**

  Si el índice del nodo más pequeño ha cambiado (ya sea el nodo actual, el hijo izquierdo o el hijo derecho), significa que uno de los hijos es más pequeño que el nodo actual. Entonces, intercambiamos el valor del nodo actual con el hijo más pequeño y llamamos recursivamente a min_heapify en el índice del hijo donde se realizó el intercambio.

  if smallest_index != index: self.data[smallest_index], self.data[index] = self.data[index], self.data[smallest_index] self.min_heapify(smallest_index)

- **Finalización:**

  La función termina cuando el nodo actual ya no tiene hijos más pequeños o cuando alcanza un nodo hoja. En ese caso, el montículo ya está ordenado con respecto a la propiedad del montículo mínimo.

Estrategia 2

Para HeapSort con Maximal Heap:

- **max_heapify en la clase Heap:**

  La función max_heapify se asegura de que el nodo actual y sus descendientes cumplan con la propiedad de montón maximal.

  Compara el nodo actual con sus hijos, y si alguno de los hijos es mayor, intercambia el nodo actual con el hijo más grande.

  Luego, se llama recursivamente a max_heapify en el hijo intercambiado para garantizar que el montón siga siendo maximal.

- **heapSort para Maximal Heap**:

  Inicializa un montón maximal con la lista dada.

  Itera mientras el montón tenga elementos.

  En cada iteración, extrae la raíz del montón (el elemento más grande) y la agrega al resultado.

  Después de extraer la raíz, coloca el último elemento del montón en la raíz y llama a max_heapify para restaurar la propiedad de montón maximal.

Para HeapSort con Minimal Heap:

- **min_heapify en la clase Heap:**

  La función min_heapify se asegura de que el nodo actual y sus descendientes cumplan con la propiedad de montón minimal.

  Compara el nodo actual con sus hijos, y si alguno de los hijos es menor, intercambia el nodo actual con el hijo más pequeño.

  Luego, se llama recursivamente a min_heapify en el hijo intercambiado para garantizar que el montón siga siendo minimal.

- **heapSort para Minimal Heap:**

  Similar a la versión maximal, pero con min_heapify en lugar de max_heapify.

  Inicializa un montón minimal con la lista dada.

  Itera mientras el montón tenga elementos.

  En cada iteración, extrae la raíz del montón (el elemento más pequeño) y la agrega al resultado.

Después de extraer la raíz, coloca el último elemento del montón en la raíz y llama a min_heapify para restaurar la propiedad de montón minimal.

Estrategia 3

- **Inicialización de propiedades de vértices y estructuras de datos:**

  Se inicializan las propiedades de los vértices (color, distancia y padre) y se construyen las estructuras de datos necesarias (en este caso, el grafo y su representación).

- **Cola de prioridad:**

  Se utiliza la función heapq.heapify() para convertir una lista en una cola de prioridad (un montón minimal en este caso).

  La cola de prioridad está formada por tuplas (distancia, vértice) donde la distancia es la clave de ordenamiento.

- **Algoritmo de Dijkstra:**

  Se inicializan las distancias de todos los vértices como infinito y la distancia del vértice de inicio como 0.

  Se insertan las distancias de todos los vértices en la cola de prioridad.

  Mientras la cola de prioridad no esté vacía:

    Se extrae el vértice con la menor distancia utilizando heapq.heappop().

    Para cada vecino del vértice extraído, se relajan las aristas si es posible.

    Después de relajar los vértices adyacentes, se reorganiza la cola de prioridad para mantener la propiedad del montón minimal.

- **Relajación de aristas:**

  Se compara la distancia actual de un vecino con la distancia acumulada del vértice extraído más el peso de la arista que los conecta.

  Si la distancia acumulada es menor que la distancia actual del vecino, se actualiza la distancia y el padre del vecino.

- **Actualización de la cola de prioridad:**

  Después de relajar los vértices adyacentes, se reorganiza la cola de prioridad para que el vértice con la menor distancia quede en la cima.

- **Resultado:**

  Después de procesar todos los vértices, se obtienen las distancias más cortas y los padres de cada vértice, lo que proporciona el camino más corto desde el vértice de inicio hasta cada uno de los demás vértices.

Casos de prueba

Estos casos de prueba se hacen al ejecutar el código y en terminal muestra la correspondiente salida y sin errores.

Análisis

Temporal 1

La complejidad de este algoritmo es O(logn), donde n es el número de elementos en el montón. Esto se debe a que en cada paso, la función min_heapify desciende a través del árbol binario, y en cada nivel compara y posiblemente intercambia un elemento con su hijo, dividiendo efectivamente el tamaño del problema a la mitad. Por lo tanto, el número de comparaciones e intercambios es proporcional a la altura del árbol, que es logn.

Temporal 2

- **Construcción del montón (build):**

  En la función build, se realiza una pasada desde la mitad del tamaño de la lista hasta el primer elemento (raíz), invocando la función heapify para cada índice. Esto asegura que se cumplan las propiedades del montón binario.

  La complejidad de construir el montón es O(n), donde n es el número de elementos en la lista.

- **Mantenimiento del montón (heapify):**

  En cada llamada a heapify, se desciende a través del árbol binario para restaurar las propiedades del montón. Esto implica comparar y posiblemente intercambiar un elemento con uno o ambos de sus hijos, y luego continuar con el hijo afectado hasta que se restaure el orden del montón.

  La complejidad de heapify es O(logn), donde n es el número de elementos en el montón.

- **Ordenación (heapSort):**

  En la función heapSort, se construye inicialmente un montón binario a partir de la lista de entrada (usando la función build), lo cual toma O(n).

  Luego, en cada iteración, se extrae el elemento mínimo (o máximo) del montón y se coloca al final de la lista. Esto se repite hasta que el montón esté vacío.

  La complejidad de extraer un elemento del montón es O(logn), ya que implica llamar a heapify.

Por lo tanto, la complejidad total de heapSort es O(nlogn).

Temporal 3

- **Inicialización de las propiedades de los vértices y la cola de prioridad:**

  Se inicializan las distancias de todos los vértices como infinito y la distancia del vértice de inicio como 0.

  Se insertan las distancias de todos los vértices en la cola de prioridad.

  Ambas operaciones tienen una complejidad de O(V), ya que se recorren todos los vértices una vez.

- **Iteraciones principales del algoritmo:**

  Mientras la cola de prioridad no esté vacía, se realizan operaciones de extracción y relajación.

Cada extracción de un vértice de la cola de prioridad (operación heapq.heappop()) tiene una complejidad de O(logV) debido a la reorganización del montón binario mínimo.

Para cada vértice extraído, se realiza la relajación de todas sus aristas adyacentes. Esto puede involucrar la actualización de las distancias y la reorganización de la cola de prioridad para mantener la propiedad del montón.

Dado que cada arista se relaja una vez, y cada extracción de vértice puede llevar a un número máximo de O(E) relajaciones (en el peor caso, donde cada vértice está conectado a todos los demás), el número total de relajaciones es O(E).

Por lo tanto, el tiempo total dedicado a la relajación es O((V+E) logV), ya que cada operación de relajación implica operaciones en la cola de prioridad.

Código 1

```
def min_heapify(self, index):
    left_index, right_index, smallest_index = self.left (index), self.right (index), index
    if left_index < len (self) and self.data[smallest_index] > self.data[left_index]:
        smallest_index = left_index
    if right_index < len (self) and self.data[smallest_index] > self.data[right_index]:
        smallest_index = right_index
    if smallest_index != index:
        self.data[smallest_index], self.data[index] = self.data[index], self.data[smallest_index]
        self.min_heapify (smallest_index)
```

Código 2

```
def heapSort(lst, config=True):
    result = []
    heap = Heap(lst, config)
    while len(heap) > 0:
        result.append(heap.data[0])  # Añadir la raíz actual al resultado
        last_element = heap.data.pop()  # Extraer el último elemento del montón
        if len(heap) > 0:
            heap.data[0] = last_element  # Colocar el último elemento en la raíz
            heap.heapify(0)  # Reorganizar el montón para mantener la propiedad del montón
    return result
```

```python
def main():
    lst = [Persona (uuid.uuid1 (), randint (MIN_BOUND, MAX_BOUND),uuid.uuid1()) for _ in range (SIZE)]

    print ("Lista original:")
    print ("\n".join (map (str, lst)))
    print ()

    print ("HeapSort (Maximal):")
    print ("\n".join (map (str, heapSort (lst, config=True))))
    print ()

    print ("HeapSort (Minimal):")
    print ("\n".join (map (str, heapSort (lst, config=False))))


main ()
```

Código 3

```python
    def dijkstra(self, s):
        self._buildVProps(s)
        Q = [(self.v_props[v]['distance'], v) for v in self.vertexes]
        heapq.heapify(Q)
        while len(Q) > 0:
            _, minimo_local = heapq.heappop(Q)
            for neighbor in self.getNeighbors(minimo_local):
                self.relax((minimo_local, neighbor, self._weight[(minimo_local, neighbor)]))
                # Reorganizar la cola de prioridad
                Q = [(self.v_props[v]['distance'], v) for _, v in Q]
                heapq.heapify(Q)
        return self.v_props
```

Documentación

Dentro del código.

Fuentes

Método insertado en el código compartido por el profesor sobre Grafos de caminos mínimos y montones binarios con el prototipo funcional de los métodos especificados en este laboratorio aplicado a un ejemplo de la vida cotidiana.

PROTOTIPO FUNCIONAL APLICADO A LA COTIDIANIDAD

El uso de un montón binario, ya sea en su versión maximal o minimal, es muy común en diversas aplicaciones, especialmente en la implementación de colas de prioridad, algoritmos de búsqueda de caminos mínimos como Dijkstra y en la ordenación eficiente de grandes cantidades de datos. En este ejemplo se propuso un escenario donde una empresa deportiva necesita organizar un torneo deportivo basándose en las edades de los participantes.

Supongamos que una empresa deportiva está organizando un torneo de deportes para jóvenes de diferentes edades. Para gestionar eficientemente las inscripciones y la planificación del torneo, la empresa utiliza un montón binario para ordenar a los participantes por sus edades.

CASOS DE PRUEBA

- **Generación aleatoria de datos de los participantes:**

  Se generan aleatoriamente los datos de los participantes, incluyendo su nombre, edad y deporte. Esto simula el proceso de inscripción en el torneo, donde los participantes se registran con estos datos.

- **Ordenamiento con HeapSort (Maximal):**

  Se utiliza la versión maximal del montón binario para ordenar a los participantes por edades de mayor a menor. Esto sería útil si el objetivo es seleccionar a los participantes más mayores para equipos de competición o para reconocimientos especiales.

  Por ejemplo, se podría utilizar para asignar capitanes de equipo o entrenadores de mayor experiencia a equipos de jóvenes deportistas.

- **Ordenamiento con HeapSort (Minimal):**

  Se utiliza la versión minimal del montón binario para ordenar a los participantes por edades de menor a mayor. Esto sería útil si el objetivo es asignar a los participantes a grupos de competición o actividades según su edad.

  Por ejemplo, se podría utilizar para agrupar a los participantes en categorías de edad para competiciones justas y equitativas.

- **Beneficios del uso del montón binario:**

  Eficiencia en la gestión de inscripciones: Al utilizar un montón binario, las inscripciones de los participantes se pueden organizar de manera eficiente según su edad, lo que facilita la gestión de la información del torneo.

- **Flexibilidad en la planificación del torneo:** Dependiendo de las necesidades del torneo, se puede elegir entre ordenar a los participantes de mayor a menor edad (Maximal) o de menor a mayor edad (Minimal), lo que permite una planificación más flexible y adaptada.

- **Optimización en la selección de participantes:** Al ordenar a los participantes por edades, se facilita la selección de capitanes, entrenadores o la creación de grupos de competición, lo que contribuye a una experiencia deportiva más organizada y satisfactoria para todos los involucrados.