LABORATORIO DE MONTONES BINARIOS

Presentado por:
Oscar Santiago Merino Suarez
Santiago Hurtado Martínez
Santiago Botero Garcia

Profesor:
Sebastián Camilo Martínez Reyes

Escuela Colombiana de Ingeniería Julio Garavito
2024-01

Introducción

Este proyecto se centra en hacer más eficiente el algoritmo de Dijkstra, que se utiliza para encontrar el camino más corto en un grafo con pesos en sus aristas. Para lograrlo, primero nos aseguramos de que nuestra estructura de datos llamada montón binario, que es clave para el algoritmo, esté optimizada. Para ello, completamos una función llamada min_heapify. Luego, adaptamos una técnica de ordenamiento llamada HeapSort para trabajar con nuestro montón binario, tanto en su versión mínima como en su versión máxima.

El montón binario es como una lista organizada de modo que siempre el elemento más pequeño esté en la parte superior. Esto es útil para el algoritmo de Dijkstra, ya que nos permite encontrar rápidamente el vértice con la distancia mínima en cada paso.

HeapSort es un método de ordenamiento muy eficiente que también aprovechamos en este proyecto. Adaptamos HeapSort para trabajar tanto con el montón binario mínimo como con el máximo, lo que nos permite ordenar nuestros datos de manera rápida y efectiva.

Finalmente, integramos esta estructura de datos optimizada en el algoritmo de Dijkstra. Así, en lugar de usar una función que extrae el mínimo, usamos nuestro montón binario, lo que hace que el algoritmo sea mucho más rápido.

Problemas

- Implementar el montón binario minimal (Completando la función min_heapify)
- Implementar las funciones de HeapSort para ambas versiones de montón binario (Maximal y minimal)
- Implementar el algoritmo de dijkstra en donde en lugar de la función extrac_min se utilice la cola de prioridad para determinar aqueel vértice con menor distancia.
- Adjuntar casos de prueba del prototipo en el informe.

Entrada

Las entradas de este problema son:

- Una lista de tipo ‘Persona’, la cual representa los vértices del grafo, donde cada persona tiene su nombre y edad.
- Una lista de tuplas que representa las aristas del grafo dirigido, es decir, las relaciones. Cada tupla contiene el nombre de el vértice de inicio, el nombre de el vértice al cual está dirigido y el peso de esa relación o, pero de la arista.

Salida

Las salidas de este problema son las siguientes:

- La representación de la matriz de adyacencia del grafo
- La representación de la lista de adyacencia del grafo
- El resultado de aplicar Dijkstra al grafo desde un vértice dado.
  Este resultado tiene la distancia más corta desde el vértice de origen a todos los vértices del grafo.
  Además de la distancia más corta del grafo a cada uno de los vértices.

Casos de Prueba

Entrada	Justificación	Salida
‘Persona 0’, ‘persona 1’, 4	Caso común	4
‘Persona 1’, ‘persona 1’, 1	Caso relación de un vértice a el mismo	Persona 1 ["('Persona 1', 1)"]
‘Persona 1’, ‘Persona 0’, 0	Caso distancia nula	0

Grafos dijkstra:

```
relaciones = [
    ('Persona 0', 'Persona 1', 4),
    ('Persona 0', 'Persona 2', 1),
    ('Persona 1', 'Persona 3', 1),
    ('Persona 2', 'Persona 1', 2),
    ('Persona 2', 'Persona 3', 5),
    ('Persona 3', 'Persona 4', 3)
]
```

Códigos

```
import math
import uuid
from random import randint


class Heap:
    # config : True // Max_Heap
    # config : False // Min_heap
    def __init__(self, data=[], config=True):
        self.data = []
        self.config = config
        self.build(data[:])

    def left(self, index):
        return 2 * index + 1

    def right(self, index):
        return 2 * (index + 1)

    def parent(self, index):
        return (index - 1) // 2

    def height(self):
        return math.ceil(math.log(len(self.data), 2))

    def __len__(self):
        return len(self.data)

    def insert(self, new):
        self.data.append(new)
        self.build()

    def update(self, old, new):
        self.delete(old)
        self.insert(new)

    def delete(self, to_delete):
        if len(self) == 0:
            raise Exception("El montón está vacio")
```

```
        if to_delete not in self.data:
            raise Exception("El elemento no está en el montón")
        self.data.remove(to_delete)
        self.build()

    def build(self, data=[]):
        if data and len(data) > 0 and isinstance(data, list):
            self.data = data
        for index in range(len(self) // 2, -1, -1):
            self.heapify(index)

    def heapify(self, index):
        if self.config == True:
            self.max_heapify(index)
        else:
            self.min_heapify(index)

    def max_heapify(self, index):
        left_index, right_index, largest_index = self.left(index),
self.right(index), index
        if left_index < len(self) and self.data[largest_index] <
self.data[left_index]:
            largest_index = left_index
        if right_index < len(self) and self.data[largest_index] <
self.data[right_index]:
            largest_index = right_index
        if largest_index != index:
            self.data[largest_index], self.data[index] = self.data[index],
self.data[largest_index]
            self.max_heapify(largest_index)

    def peek(self):
        return self.data[0]

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

```
            self.max_heapify(largest_index)

    def __str__(self):
        return str(self.data)


class PriorityQueue:
    def __init__(self, data=[], config=True):
        self.data = Heap(data, config)

    def __str__(self):
        return str(self.data)

    def __len__(self):
        return len(self.data)

    def enqueue(self, new):
        self.data.insert(new)

    def dequeue(self):
        if len(self) == 0:
            raise Exception("Underflow")
        to_dequeue = self.data.peek()
        self.data.delete(to_dequeue)
        return to_dequeue

    def update_priority(self, old, new):
        self.data.update(old, new)


class Persona:
    def __init__(self, nombre="", edad=1):
        self.nombre = nombre
        self.edad = edad

    def __str__(self):
        return str({
            "Nombre": self.nombre,
            "Edad": self.edad
        })

    def __lt__(self, other):
        return self.edad < other.edad


MAX_BOUND = 72
```

```
MIN_BOUND = 18
SIZE = 10

def heapSort(lst, config=True):
    result = []
    pq = PriorityQueue(lst, config)
    while len(pq) > 0:
        result.insert(0, pq.dequeue())
    return result

def main():
    lst = [Persona(uuid.uuid1(), randint(MIN_BOUND, MAX_BOUND)) for e in
range(SIZE)]
    pq = PriorityQueue(lst,True)
    print("Se usa el min_heapify con False y max_heapify con True")
    while len(pq) > 0:
        print('Atendiendo al cliente con edad ... ', pq.dequeue())


    print("Lista ordenada usando HeapSort (Heap máximo):")
    sorted_lst = heapSort(lst, True)
    print(list(map(str, sorted_lst)))
    print("Lista ordenada usando HeapSort (Heap mínimo):")
    sorted_lst = heapSort(lst, False)
    print(list(map(str, sorted_lst)))


main()
```

```
from collections import deque
import math
from random import randint

WHITE = "white"
BLACK = "black"
GRAY = "gray"

class Graph:
    def _buildAdjMatrix(self):
        self.adjMat = [[0 for v in range(len(self.vertexes))] for v in
range(len(self.vertexes))]
        for relation in self.relations:
            row, col = self.encoder[relation[0]], self.encoder[relation[1]]
            self.adjMat[row][col] = relation[2]  # Usar el peso en la matriz
de adyacencia

    def _buildEncoding(self):
        self.encoder, self.decoder = {}, {}
        index = 0
        for v in self.vertexes:
            self.encoder[v] = index
            self.decoder[index] = v
            index = index + 1

    def _buildAdjList(self):
        self.adjList = {}
        for v in self.vertexes:
            self.adjList[v] = []
        for relation in self.relations:
            self.adjList[relation[0]].append((relation[1], relation[2]))  #
Guardar el peso junto con el vecino

    def _buildRelation(self, e):
        if self.directed:
            self.relations = e
        else:
            self.relations = set()
            for el in e:
                self.relations.add(el)
                if len(el) == 3:
                    self.relations.add((el[1], el[0], el[2]))

    def __init__(self, v, e, directed=True, view=True):
        self.directed = directed
        self.view = view
```

```
        self.vertexes = v
        self._buildRelation(e)
        self._buildAdjList()
        self._buildEncoding()
        self._buildAdjMatrix()
        self._buildWeight()

    def _buildWeight(self):
        self._weight = {}
        for relation in self.relations:
            self._weight[(relation[0], relation[1])] = relation[2]

    def getAdjMatrix(self):
        return self.adjMat

    def getAdjList(self):
        return self.adjList

    def dijkstraP(self, s):
        self._buildVProps(s)
        QS = PriorityQueue([(self.v_props[v]['distance'], v) for v in
self.vertexes], config=False)  # Min-Heap
        in_queue = {v: True for v in self.vertexes}  # Track vertices in the
queue
        while len(QS) > 0:
            u = QS.dequeue()[1]
            in_queue[u] = False  # Mark as dequeued
            for neighbor, weight in self.getNeighbors(u):
                if self.v_props[neighbor]['distance'] >
self.v_props[u]['distance'] + weight:
                    old_distance = self.v_props[neighbor]['distance']
                    self.v_props[neighbor]['distance'] =
self.v_props[u]['distance'] + weight
                    self.v_props[neighbor]['parent'] = u
                    if in_queue[neighbor]:
                        QS.update((old_distance, neighbor),
(self.v_props[neighbor]['distance'], neighbor))
        return self.v_props

    def _buildVProps(self, source=None):
        self.v_props = {}
        for v in self.vertexes:
            self.v_props[v] = {
                'color': WHITE,
                'distance': math.inf,
                'parent': None
```

```
            }
        if source is not None:
            self.v_props[source]['distance'] = 0

    def _getNeighborsAdjList(self, vertex):
        return self.adjList[vertex]

    def getNeighbors(self, vertex):
        return self._getNeighborsAdjList(vertex)

    def bfs(self, source):
        self._buildVProps(source)
        queue = [source]
        while len(queue) > 0:
            u = queue.pop(0)
            for neighbor, _ in self.getNeighbors(u):
                if self.v_props[neighbor]['color'] == WHITE:
                    self.v_props[neighbor]['color'] = GRAY
                    self.v_props[neighbor]['distance'] =
self.v_props[u]['distance'] + 1
                    self.v_props[neighbor]['parent'] = u
                    queue.append(neighbor)
            self.v_props[u]['color'] = BLACK
        return self.v_props

    def dfs(self):
        self._buildVProps()
        time = 0
        for v in self.vertexes:
            if self.v_props[v]['color'] == WHITE:
                time = self.dfs_visit(v, time)
        return self.v_props

    def dfs_visit(self, vertex, time):
        time = time + 1
        self.v_props[vertex]['distance'] = time
        self.v_props[vertex]['color'] = GRAY
        for neighbor, _ in self.getNeighbors(vertex):
            if self.v_props[neighbor]['color'] == WHITE:
                self.v_props[neighbor]['parent'] = vertex
                time = self.dfs_visit(neighbor, time)
        self.v_props[vertex]['color'] = BLACK
        time = time + 1
        self.v_props[vertex]['final'] = time
        return time
```

```
def printVProps(v_props):
    print("====================== Results ======================")
    for v in v_props.keys():
        v_props[v]['path'] = '-->'.join(map(str, getPath(v, v_props)))
        print(str(v), '-->', v_props[v])

def printAdjMatrix(graph):
    print("====================== ADJ Matrix ======================")
    adjMat = graph.getAdjMatrix()
    for row in adjMat:
        print(' '.join(list(map(str, row))))

def getPath(vertex, v_props):
    path = [vertex]
    current = vertex
    while (v_props[current]['parent'] is not None):
        path.insert(0, v_props[current]['parent'])
        current = v_props[current]['parent']
    return path

def printAdjList(graph):
    print("====================== ADJ List ======================")
    adjList = graph.getAdjList()
    for v in adjList.keys():
        print(str(v), list(map(str, adjList[v])))

class Heap:
    def __init__(self, data=[], config=True):
        self.data = []
        self.config = config
        self.build(data[:])

    def left(self, index):
        return 2 * index + 1

    def right(self, index):
        return 2 * (index + 1)

    def parent(self, index):
        return (index - 1) // 2

    def height(self):
        return math.ceil(math.log(len(self.data), 2))

    def __len__(self):
        return len(self.data)
```

```
    def insert(self, new):
        self.data.append(new)
        self.build()

    def update(self, old, new):
        try:
            index = self.data.index(old)
            self.data[index] = new
            self.heapify(index)
        except ValueError:
            # Si old no se encuentra en self.data, no hacemos nada
            pass

    def delete(self, to_delete):
        if len(self) == 0:
            raise Exception("El montón está vacio")
        if to_delete not in self.data:
            raise Exception("El elemento no está en el montón")
        self.data.remove(to_delete)
        self.build()

    def build(self, data=[]):
        if data and len(data) > 0 and isinstance(data, list):
            self.data = data
        for index in range(len(self) // 2, -1, -1):
            self.heapify(index)

    def heapify(self, index):
        if self.config:
            self.max_heapify(index)
        else:
            self.min_heapify(index)

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

```
            self.max_heapify(largest_index)

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

    def peek(self):
        return self.data[0]

    def __str__(self):
        return str(self.data)

class PriorityQueue:
    def __init__(self, data=[], config=True):
        self.data = Heap(data, config)

    def __str__(self):
        return str(self.data)

    def __len__(self):
        return len(self.data)

    def enqueue(self, new):
        self.data.insert(new)

    def dequeue(self):
        if len(self) == 0:
            raise Exception("Underflow")
        to_dequeue = self.data.peek()
        self.data.delete(to_dequeue)
        return to_dequeue

    def update(self, old, new):
        self.data.update(old, new)

class Persona:
```

```
    def __init__(self, nombre="", edad=1):
        self.nombre = nombre
        self.edad = edad

    def __str__(self):
        return str({
            "Nombre": self.nombre,
            "Edad": self.edad
        })

    def __lt__(self, other):
        return self.edad < other.edad

MAX_BOUND = 72
MIN_BOUND = 18
SIZE = 10

def heapSort(lst, config=True):
    heap = Heap(lst, config)
    result = []
    while len(heap) > 0:
        result.append(heap.peek())
        heap.delete(heap.peek())
    return result

def main():
    lst = [Persona(f'Persona {i}', randint(MIN_BOUND, MAX_BOUND)) for i in
range(SIZE)]

    vertices = [persona.nombre for persona in lst]
    relaciones = [
        ('Persona 0', 'Persona 1', 4),
        ('Persona 0', 'Persona 2', 1),
        ('Persona 1', 'Persona 3', 1),
        ('Persona 2', 'Persona 1', 2),
        ('Persona 2', 'Persona 3', 5),
        ('Persona 3', 'Persona 4', 3)
    ]

    graph = Graph(vertices, relaciones, directed=True, view=True)

    printAdjMatrix(graph)
    printAdjList(graph)

    source_vertex = 'Persona 0'
    print("====================== Dijkstra con cola de prioridad
```

```
====================")
    dijkstra_result = graph.dijkstraP(source_vertex)
    printVProps(dijkstra_result)

main()
```