Algoritmi e Strutture Dati Laboratorio

Anno accademico 2021/2022

Progetto totale 1

Carmine Marchesani
Matricola: 113916

Patryk Sebastian Bialowas:
Matricola: 113959

Obiettivi del progetto:

- Realizzare in Java una implementazione di un **albero binario di ricerca AVL**.
- Realizzare in Java una implementazione di un algoritmo di **ordinamento basato su alberi AVL**, dell'algoritmo di ordinamento **HeapSort basato su uno heap ternario** e dell'algoritmo di ordinamento **CountingSort**.
- **Valutare numericamente le prestazioni** degli algoritmi di ordinamento implementati al punto precedente.
- Realizzare in Java una implementazione di un **grafo generico orientat**o con matrice di adiacenza.
- Per il problema dei cammini minimi con sorgente singola, realizzare in Java una implementazione dell'**algoritmo di Bellman-Ford**.
- Per il problema dei cammini minimi tra tutte le coppie di nodi, realizzare in Java una implementazione dell'algoritmo di **Floyd-Warshall** utilizzando la **programmazione dinamica**.

ALBERI AVL

Un Albero AVL è un albero binario di ricerca che si mantiene sempre **bilanciato**.
Il coefficiente di bilanciamento di per ciascun nodo dell'albero è compreso nell'intervallo [-1, 1].
Per garantire questo bilanciamento vengono effettuate delle rotazioni (Destra-Destra, Sinistra-Sinistra, Destra-Sinistra o Sinistra-Destra).
Il momento saliente di quest' implementazione è l'inserimento dell'elemento nell' AVL Tree.

Segue l'implementazione dell'inserimento.

```
public int insert(E el) {
    int count = 0;
    AVLTreeNode tmp = this;
    while(true)
    {
        if(el.compareTo(tmp.getEl()) == 0)
        {
            tmp.setCount(tmp.getCount() + 1);
            return -1;
        }
        else if (el.compareTo(tmp.getEl()) < 0)
        {
            count++;
            if(tmp.getLeft() == null)
            {
                tmp.setLeft(new AVLTreeNode(el, tmp));
                tmp=tmp.getLeft();
                //return count;
                break;
            }

            else
                tmp = tmp.getLeft();
        }
        else if(el.compareTo(tmp.getEl()) > 0)
        {
            count++;
            if(tmp.getRight() == null)
            {
                tmp.setRight(new AVLTreeNode(el, tmp));
                //return count;
                tmp = tmp.getRight();
                break;
            }
            else
                tmp = tmp.getRight();
        }
    }
    this.scanningTree(tmp);
    this.rebalance(tmp);
    return count;
```

AVLTree Sort

L'algoritmo AVLTree Sort consiste nell'ordinare una lista di elementi. Una volta inseriti gli elementi si effettua una visita in-order di tutto l'albero AVL. Il metodo **`inOrderVisit()`** sarà implementato in AVLTree.

Segue l'implementazione di AVLTree Sort.

```
public SortingAlgorithmResult<E> sort(List<E> l) {
    //creo un avl tree vuoto
    AVLTree<E> avlTree= new AVLTree<>();
    int compareCounter = 0;
    /*
    * inserisco nell'avl tutti gli elementi della lista
    * e accumulo il numero di confronti
    */
    for (E element: l)
        compareCounter += avlTree.insert(element);

    //ritorno il numero di confronti e lista ottenuta
    return new SortingAlgorithmResult<>(avlTree.inOrderVisit(), compareCounter);
}
```

L'aggiunta di un elemento a un albero di ricerca binaria richiede in **media O(log n)** di complessità temporale.
Pertanto, l'aggiunta di n elementi richiede **O(n log n)** di complessità temporale.
Nel caso peggiore la complessità temporale sarà **O(n ^ 2)**.

Heap sort

La Heapsort, per eseguire l'ordinamento, utilizza una struttura chiamata heap; un heap è rappresentabile con un albero binario in cui tutti i nodi seguono una data proprietà.
I momenti salienti dell'implementazione di questa classe sono stati la procedura di **heapify** e il build **Max Heap.**

```
private void buildMaxHeap(List<E> l)
{
    for (int i = l.size() / 3; i >= 0; i--)
        heapify(l, i);
}
```

La complessità costa all'esecuzione un **O(n)**. Si occupa di costruire il **Max Heap** prima di passare ad ordinare l' heap, dove ogni parente e' più grande dei suoi figli.

```
private void heapify(List<E> l, int i) {
    int max = i;
    if(this.getFirstChildNode(i) < dim && l.get(this.getFirstChildNode(i)).compareTo(l.get(max)) >= 0)
    {
        max = this.getFirstChildNode(i);
    }
    if(this.getSecondChildNode(i) < dim && l.get(this.getSecondChildNode(i)).compareTo(l.get(max)) >= 0)
    {
        max = this.getSecondChildNode(i);
    }
    if(this.getThirdChildNode(i) < dim && l.get(this.getThirdChildNode(i)).compareTo(l.get(max)) >= 0)
    {
        max = this.getThirdChildNode(i);
    }
    if(max != i)
    {
        swap(l, i, max);
        heapify(l, max);
        count++;
    }
}
```

Mentre **heapify** che si occupa di ricostruire il maxheap e portare con i vari swap l'elemento piu' grande in cima detiene una complessita' **O(log i)** dove **i** rappresenta il numero di elementi

$$T(n) = \sum_{i=2}^{n} \log i + \Theta(n) = \Theta(n \log n)$$

Nel complesso per calcolare la complessità di un HeapSort si effettua il calcolo sopracitato.

Counting sort

Counting sort è un algoritmo di ordinamento che ordina gli elementi di un array contando il numero di occorrenze di ciascun elemento univoco nell'array. Il conteggio viene archiviato in un array ausiliario e l'ordinamento viene eseguito mappando il conteggio come indice dell'array ausiliario.

Counting sort è anche detto come algoritmo di ordinamento stabile, perché mantiene in esatto ordine anche in caso di omonimia.

STEP 1

Per iniziare ad ordinare l'array ci serve conoscere il range di valori che andremo ad ordinare, nel nostro caso trovo il valore minimo e massimo.

```
for (int n : l) {
    if(n <= min)
        min = n;
    if(n >= max)
        max = n;
}
```

STEP 2

Si crea un array che servirà per tenere il conteggio di ogni elemento, le posizioni dei array corrispondono agli elementi da ordinare.

```
int[] tmpArr = new int[max-min+1];
for (int n : l)
{
    tmpArr[n-min] = tmpArr[n-min] +1;
}
```

STEP 3

Memorizza la somma cumulativa degli elementi della matrice di conteggio. Aiuta a posizionare gli elementi nell'indice corretto dell'array ordinato.

```
for (int i = 1; i < tmpArr.length ; i++)
{
    tmpArr[i] = tmpArr[i] + tmpArr[i-1];
}
```

STEP 4

Trova l'indice di ogni elemento dell'array originale nell'array count. Questo dà il conteggio cumulativo. Posizionando l'elemento all'indice calcolato come mostrato nella figura seguente.

```
for(int i = result.size()-1; i >= 0; i--)
{
    result.set(tmpArr[l.get(i)-min]-1, l.get(i));
    tmpArr[l.get(i)-min] = tmpArr[l.get(i)-min] -1;
}
```

Complessita'

La complessita' temporale di questo algoritmo e' sempre la stessa **O(n+k)** in quanto l'algoritmo riscrive completamente l'array ordinato.
I vari cicli for possono avere solo 2 tipi di complessità:

O(max) grandezza range
O(size) lunghezza del array

`

$$T(n)= O(k) + O(n) = O(k+n)$$

Grafo generico orientato

Un grafo orientato è una struttura matematica che consiste in un insieme di archi e vertici.
Il grafo orientato tramite gli archi collega due vertici in modo asimmetrico.
Un Grafo Orientato è rappresentato dalla coppia **G=(V, E)** , dove **V** è l'insieme di vertici ed **E** è un insieme di archi orientati che collega le varie coppie di vertici.

Adjacency Matrix Directed Graph

La classe che segue implementa un grafo orientato mediante matrice di adiacenza
L'insieme di nodi con il loro relativo indice nella matrice di adiacenza verrà rappresentata tramite una Map.

```
protected Map<GraphNode<L>, Integer> nodesIndex;
```

Per permettere alla matrice di espandersi in modo **regolare** in base all'inserimento dei nodi verrà utilizzata la struttura dati dell' ArrayList. In quest' implementazione la matrice è rappresentata da un ArrayList di ArrayList contenente gli archi del grafo orientato. Tutto questo combinato con la Map permette di aggiungere, rimuovere e cercare elementi.

```
protected ArrayList<ArrayList<GraphEdge<L>>> matrix;
```

I momenti salienti dell'implementazione di questa struttura dati sono stati l'aggiunta e la rimozione degli archi.

Segue l'implementazione di aggiunta di un arco al grafo.

```
public boolean addEdge(GraphEdge<L> edge) {
    if(edge == null)
        throw new NullPointerException("L'arco passato è nullo");

    if(!(nodesIndex.containsKey(edge.getNode1()) && nodesIndex.containsKey(edge.getNode2())))
        throw new IllegalArgumentException("Uno dei due nodi specificati nell'arco non esiste");

    if(!edge.isDirected())
        throw new IllegalArgumentException("Questo arco e' non orientato");

    //scorro tramite foreach tutti gli array list dentro a matrix
    for(ArrayList<GraphEdge<L>> graphEdges : matrix) {
        for (GraphEdge<L> edgeApp : graphEdges)
        {
            if(edge.equals(edgeApp))
                return false;
        }
    }

    matrix.get(this.getNodeIndexOf(edge.getNode1())).add(edge);
    return true;
}
```

Segue l'implementazione per la rimozione di un arco da un grafo.

```
public void removeEdge(GraphEdge<L> edge) {
    if(edge == null)
        throw new NullPointerException("L'arco e' nullo");
    if(this.getEdge(edge) == null)
        throw new IllegalArgumentException("L'arco non esiste in questo grafo");
    if(!nodesIndex.containsKey(edge.getNode2()) || !nodesIndex.containsKey(edge.getNode1()))
        throw new IllegalArgumentException("Un nodo e' nullo");
    // toChange assume il valore del arraylist della posizione specifica di node1.
    ArrayList<GraphEdge<L>> toChange = matrix.get(nodesIndex.get(edge.getNode1()));
    //rimuovo l'arco da toChange
    toChange.remove(edge);
    //Sovrascrivo la riga di matrix con toChange
    matrix.set(nodesIndex.get(edge.getNode1()), toChange);
}
```

Algoritmi per il cammino minimo

Il problema del cammino minimo consiste nel trovare il cammino tra due vertici di un grafo tale che la somma degli archi che costituiscono il cammino sia minima.
Esistono vari algoritmi per il calcolo del cammino minimo. Nelle delucidazioni che seguono verranno mostrati alcuni di essi.

Algoritmo di Bellman Ford

L'algoritmo di Bellman Ford risolve il problema dei cammini minimi tra nodi di un grafo da sorgente unica. In quest' algoritmo possono essere presenti pesi negativi, ma non cicli di peso negativo.
In caso di ciclo negativo, il problema non ha soluzione, poiché il ciclo verrebbe ripetuto all'infinito, altrimenti questo algoritmo restituisce i cammini minimi e i loro pesi.

Assumendo che i controlli e le operazioni aritmetiche siano svolte in tempo costante $O(1)$, la complessità temporale dell'algoritmo è:
$O(N + N \cdot M + M) \approx O(N \cdot M)$
dove $N = |V|$ e $M = |E|$.

Segue l'implementazione che controlla il caso di ciclo negativo.

```
for (GraphEdge<L> newEdge: this.getGraph().getEdges())
    if((newEdge.getNode1().getFloatingPointDistance() + newEdge.getWeight()) <
            newEdge.getNode2().getFloatingPointDistance())
        throw new IllegalStateException("Il grafo presenta un ciclo con peso negativo.");
```

Segue l'implementazione che esegue il cammino minimo da un nodo sorgente.

```java
public void computeShortestPathsFrom(GraphNode<L> sourceNode)
{
    if(sourceNode == null)
        throw new NullPointerException("Il nodo passato e' nullo.");

    if(!graph.getNodes().contains(sourceNode))
        throw new IllegalArgumentException("Il nodo passato non esiste.");

    //step 1
    //confronto ogni nodo del grafo con un nuovo nodo
    //source sarà il nodo da scoprire
    for (GraphNode<L> next : this.getGraph().getNodes())
    {
        //il nuovo nodo sorgente avrà distanza pari a infinito
        next.setFloatingPointDistance(Double.POSITIVE_INFINITY);
        next.setPrevious(null);
        //se il nodo passato è uguale a quello da confrontare
        //vuol dire che quello sarà a prescindere il nodo
        //di partenza da cui inizia a calcolare il
        //cammino minimo, quindi lo setto a 0.
        if(sourceNode.equals(next))
            next.setFloatingPointDistance(0.0);
    }
    //scorro tutti i nodi del grafo
    for (int i = 0; i < this.getGraph().nodeCount() - 1; i++)
    {
        //confronto gli archi e i loro pesi
        for (GraphEdge<L> newEdge : this.graph.getEdges())
        {
            if (newEdge.getNode2().getFloatingPointDistance() > newEdge.getNode1().getFloatingPointDistance()
                    + newEdge.getWeight())
            {
                newEdge.getNode2().setFloatingPointDistance(newEdge.getNode1().getFloatingPointDistance()
                        + newEdge.getWeight());
                newEdge.getNode2().setPrevious(newEdge.getNode1());
            }
        }
    }
   //controllo la presenza di un ciclo negativo
    for (GraphEdge<L> newEdge: this.getGraph().getEdges())
        if((newEdge.getNode1().getFloatingPointDistance() + newEdge.getWeight()) <
                newEdge.getNode2().getFloatingPointDistance())
            throw new IllegalStateException("Il grafo presenta un ciclo con peso negativo.");

    this.isComputed = true;
    this.lastSource = sourceNode;
}
```

Algoritmo di Floyd Warshall

L'algoritmo di Floyd-Warshall è un algoritmo per trovare il percorso più breve tra tutte le coppie di vertici in un grafo pesato. Questo algoritmo funziona sia per i grafici ponderati diretti che non orientati. Ma non funziona per i grafici con cicli negativi (dove la somma degli archi in un ciclo è negativa).

Nel cuore dell'algoritmo ci sono 3 cicli che sono identici tra di loro **O(V)** dove V e' il numero di vertici.
Visto che e' un algoritmo di programmazione dinamica calcolare la complessità temporale risulta

$$O(V^3)$$

```java
public void computeShortestPaths()
{
    //inserisco nella matrice i valori con archi diretti al nodo sorgente
    for (GraphEdge<L> edge : this.getGraph().getEdges()) {
        this.getCostMatrix()[this.getGraph().getNodeIndexOf(edge.getNode1().getLabel())]
                            [this.getGraph().getNodeIndexOf(edge.getNode2().getLabel())]
                            = edge.getWeight();
        this.getPredecessorMatrix()[this.getGraph().getNodeIndexOf(edge.getNode1().getLabel())]
                                   [this.getGraph().getNodeIndexOf(edge.getNode2().getLabel())]
                                    = this.getGraph().getNodeIndexOf(edge.getNode1().getLabel());
    }

    //costruisco e inserisco in contemporanea come specificato dal algoritmo di Floyd
    //la matrice costMatrix e PredecessorMatrix
    for (int k = 1; k < this.getGraph().nodeCount(); k++)
        for(int i = 1; i < this.getGraph().nodeCount(); i++)
            for (int j = 1; j < this.getGraph().nodeCount(); j++)
                if(this.getCostMatrix()[i][j] > this.getCostMatrix()[i][k] + this.getCostMatrix()[k][j])
                {
                    this.getCostMatrix()[i][j] = this.getCostMatrix()[i][k] + this.getCostMatrix()[k][j];
                    this.getPredecessorMatrix()[i][j] = this.getPredecessorMatrix()[k][j];
                }

    //faccio la somma dei pesi del grafo, se il peso risulta negativo
    //scatta illegalStateException()
    double integer = 0;
    for (GraphEdge<L> edge : this.getGraph().getEdges()) {
        integer = integer + edge.getWeight();
    }
    if(integer<0)
        throw new IllegalStateException("Il peso e' negativo");

    this.solved = true;
}
```

Test

Per verificare la correttezza delle classi create sono stati implementati dei test JUnit.
JUnit è un framework per unit testing, cioè permette di effettuare test su singole sezioni di codice.
In riferimento la repository GitHub del progetto, contenente le classi e i relativi file di test:
https://github.com/MrEntity303/ST0853-ASD