TDA416 - Lab1.2

Grupp 46
Pontus Lindblom, Stefan Chan

November 23, 2018

1.2

1.2A

Algoritmanalys

Algoritmen lyder:

1: Beräkna värdet av varje punkt och lagra resultatet i fält
2: while antalet punkter ¡ k
3: tag bort den minst betydelsefulla
4: beräkna om dess tidigare grannars värde (2st)
end while

Rad för rad analys ger följande grov uppskattning:

1: Tar n (total antal punkter) operationer om beräkning av värdet räknas som en operation då den operationen har maximal komplexitet $\mathcal{O}(n)$.
2: Loopen och dess innehåll kommer att köras $(n-k)$ antal gånger, här görs en operation.
3: Vi antar här att bortagningen är av komplexitet $\mathcal{O}(n)$, men kan vara mycket dyrare.
4: Beräkningen räknas till en operation per granne.

Detta ger att vår algoritm får följande komplexitet:

$$\sum_{i=0}^{n} 1 + \sum_{k=0}^{n-k} (1+n+2)$$

Vilket beräknas till:

$$\sum_{i=0}^{n} 1 + \sum_{k=0}^{n-k} (1+n+2) = n+1+(\sum_{k=0}^{n-k} 3 + \sum_{k=0}^{n-k} n)$$

De inre summorna efter faktorisering beräknas till:

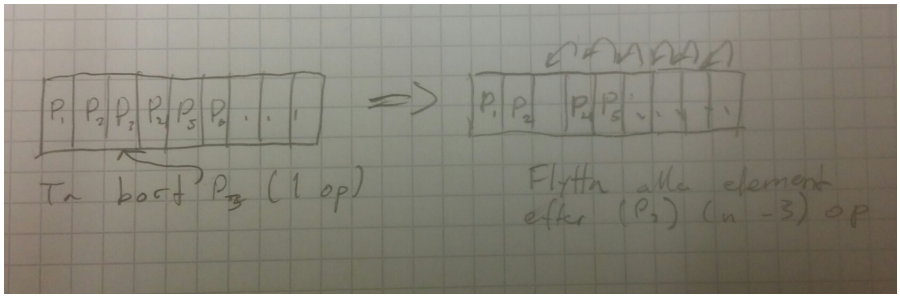
$$3\sum_{k=0}^{n-k} 1 + \sum_{k=0}^{n-k} n = 3(n-k+1) + n(n-k+1)$$

Komplexiteten blir:

$$n+1+3(n-k+1)+n(n-k+1) = n^2+4n-kn-3k+4 \in \mathcal{O}(n^2)$$

Med vilkoret att handlingar av $\mathcal{O}(n)$ räknas som n antal operationer och att k är en valfri konstant. Detta är dock en väldig grov underskattning och är förmodligen mycket mer dyrare.

Varför fält tar lång tid



Bortagning av element kan bli dyrt beroende på storleken av fältet och vart elementatet man tar bort ligger i fältet. Förutom att ta bort elementet måste alla andra element efter flyttas ner en index, vilket ger hela operationen innefattar (size - index) andra operationer.

Varför LinkedList är också dålig

LinkedList kräver att man traverserar från början av listan ifall man behöver tillgång till en viss element i listan. Detta är speciellt märkbart ifall man måste iterera flera gånger som vår algortim gör.

1.2B

Metod addLast:

```
    public void addLast(Point p) {
    if (p == null){
       throw new NullPointerException("addLast: The Point to add is
           null");
    }

    Node lastNode = new Node(p, -1);

    if (head == null){
       head = lastNode;
       tail = head;
    } else {

       tail.next = lastNode;
       lastNode.prev = tail;
       tail = lastNode;

    }

  }
```

Metod reduceListToKElements:

```
    public void reduceListToKElements(int k) {
    // The current node when traversing the double linked list
    Node current = head.next;

    // Gives an importance value to all nodes (except the first and
        the last) and places them in the priority queue
    while (current.next != null && current.prev != null) {
       current.imp = importanceOfP(current.prev.p, current.p,
           current.next.p);
       q.add(current);
       size++;
       current = current.next;
    }
    // Reduces the list to the k most important nodes, accounting for
        the first and the last nodes
    while (q.size() > k) {
       // Catches exceptions when the list is empty.
       try {
          Node lowValueNode = q.poll();
          lowValueNode.prev.next = lowValueNode.next;
          lowValueNode.next.prev = lowValueNode.prev;
          size--;
```

```
            calculateNeighbourImp(lowValueNode.prev);
            q.remove(lowValueNode.prev);
            q.offer(lowValueNode.prev);           // Updates the priority
                queue when a new node is inserted
            calculateNeighbourImp(lowValueNode.next);
            q.remove(lowValueNode.next);       // Update the priority
                queue
            q.offer(lowValueNode.next);
        } catch (Exception ex) {
            throw new NoSuchElementException("reduceListToKElements: The
                priority queue is empty");
        }
    }
}
```

PriorityQueue uppdateras bara när ett nytt element läggs in i kön, därför tar vi ut och lägger tillbaka noderna när de uppdateras så att PriorityQueue sorterar de uppdaterade noderna.

1.2C

Vår EO: Tilldelningar utanför loopar och access generellt är gratis.

```
public void reduceListToKElements(int k) {
    Node current = head.next; // 1 op

    while (current.next != null && current.prev != null) { // 2n op
        current.imp = importanceOfP(current.prev.p, current.p,
            current.next.p); // Assume 1 op for n loops
        q.offer(current); // log(n) op for n loops
        size++; // 2 op for n loops
        current = current.next; // 1 op for n loops
    }

    while (q.size() > k) { // 1 op for (n - k) loops
        try {
            Node lowValueNode = q.poll(); // log(n) op for (n - k) loops
            lowValueNode.prev.next = lowValueNode.next; // 1 op for (n -
                k) loops
            lowValueNode.next.prev = lowValueNode.prev; // 1 op for (n -
                k) loops
            size--; // 2 op for (n - k) loops

            calculateNeighbourImp(lowValueNode.prev); // Assume 1 op for
                (n - k) loops
            q.remove(lowValueNode.prev); // n op for (n - k)
            q.offer(lowValueNode.prev); // log(n) op for (n - k) loops
```

```
            calculateNeighbourImp(lowValueNode.next); // Assume 1 op for
                (n - k) loops
            q.remove(lowValueNode.next); // n op for (n - k)
            q.offer(lowValueNode.next); // log(n) op for (n - k) loops
        } catch (Exception ex) {// Should not happen if run correctly
            throw new NoSuchElementException("reduceListToKElements: The
                priority queue is empty");
        }
    }
}
```

Summering utav alla operationer ger:

$$1 + \sum_{i=1}^{n}(1 + 2 + log(n) + 2 + 1) + \sum_{k=1}^{n-k}(log(n) + 1 + 1 + 2 + n)$$

Tilldelning, konsolidering av termer och eventuella faktoriseringar ger:

$$1 + 6\sum_{i=1}^{n} 1 + \sum_{i=1}^{n} log(n) + 3\sum_{k=1}^{n-k} log(n) + 2\sum_{k=1}^{n-k} n + 5\sum_{k=1}^{n-k} 1$$

Summorna löses till:

$$6\sum_{i=1}^{n} 1 = 6n$$

$$\sum_{i=1}^{n} log(n) = nlog(n)$$

$$3\sum_{k=1}^{n-k} log(n) = (n-k)log(n)$$

$$2\sum_{k=1}^{n-k} n = 2n(n-k)$$

$$5\sum_{k=1}^{n-k} 1 = 5(n-k)$$

Sammanlagt blir det:

$$1 + 6n + nlog(n) + (n-k)(3log(n) + 2n + 5) \in \mathcal{O}(n^2)$$

Varför den fortfarande inte är bra

Anledningen varför algoritmen har komplexitet $\mathcal{O}(n^2)$ och inte som mest $\mathcal{O}(n)$ ligger i hur PriorityQueue uppdaterar sig själv. Den enda gången som PriorityQueue uppdaterar och sorterar innehållet är när element sätts in i listan, klassen har normalt inget sätt att uppdatera sitt innehåll. Detta gör att elementet måste tas ut från kön först för att uppdatering av innehållet och sedan läggas in tillbaka i kön. Att lägga in element är av $\mathcal{O}(log(n))$ komplexitet och inte så farligt, men att ta bort en viss element ur kön är $\mathcal{O}(n)$ komplexitet vilket är varför komplexiten av algoritmen är som den är.