Scuola di Scienze Matematiche, Fisiche e Naturali
Corso di Laurea in Informatica

Tesi di Laurea

COMPOSIZIONE DI OPERATORI DI ORDINAMENTO CON CONTENITORI

COMPOSITION OF SORTING OPERATORS WITH CONTAINERS

ALESSIO SANTORO

Relatore: *Relatore*
Correlatore: *Correlatore*

Anno Accademico 2023-2024

"Inserire citazione"
— Inserire autore citazione

1

INTRODUZIONE

2

ALGORITMI DI ODRINAMENTO E CLASSI DI PATTERN

In questo capitolo verranno introdotti i principali algoritmi di ordinamento che utilizzano sorting devices, in particolare stack-sort, queue-sort e bubble sort, e altri concetti necessari per l'analisi della loro composizione.

Durante l'esecuzione questi algoritmi possono salvare gli elementi in un contenitore (la diversa struttura dati adottata definise i diversi algoritmi) dalla quale poi vengono prelevati per essere aggiunti all'output.

Una sola iterazione non garantisce l'ordinamento della permutazione, dunque gli algoritmi devono essere iterati piú volte, ogni volta sul risultato della iterazione precedente. In ogni caso alla fine delle i-esima iterazione i maggiori i elementi avranno raggiunto la loro posizione finale, dunque sono necessari al massimo $n-1$ iterazioni per ordinare la permutazione.

Essendo interessati al comportamento di una sola iterazione di questi algoritmi si esaminerá un operatore, definito appositamente per ogni algoritmo, che descrive la singola iterazione.
Ad esempio, prendendo l'algoritmo bubble-sort si fará riferimento all'operatore $B(\pi)$, dove π é una permutazione di interi, tale che n iterazioni del bubble-sort possano essere rappresentate da $B^n(\pi) = B(\dots B(\pi)\dots)$.

BUBBLE SORT

L'algoritmo di ordinamento bubble-sort prevede di scorrere gli elementi da ordinare dal primo al penultimo, ed ogni volta confrontare ogni elemento con il suo successivo per scambiarli se non sono ordinati.
Il risultato di una singola iterazione di bubble-sort su una permutazione $\pi = \pi_1\pi_2\dots\pi_n$ é calcolato dall'operatore $B(\pi)$. Per una permutazione π

con valore massimo n vale che $\pi = \pi_L n \pi_R$, allora $B(\pi) = B(\pi_L)\pi_r n$.

Algorithm 1 $B(\pi)$

```
1: for i = 1 to n − 1 do
2:     if π_i > π_{i+1} then
3:         Swap π_i and π_{i+1}
4:     end if
5: end for
```

STACK SORT

L'operatore $S(\pi)$ rappresenta il risultato ottenuto applicando un'iterazione di stack sort su una permutazione π.
Il primo passo consiste nell'inserire π_1 nella pila. Poi lo si confronta con l'elemento π_2. Se $\pi_1 > \pi_2$ allora il secondo viene messo nella pila sopra π_1, altrimenti π_1 viene estratto dalla pila e inserito nell'output e π_2 viene inserito nella pila.
Gli stessi passi vengono eseguiti per tutti gli altri elementi presenti nell'input, se viene trovato un elemento nell'input maggiore dell'elemento in cima alla pila, la pila viene svuotata finché questa condizione non diviene falsa, poi l'elemento viene spinto nella pila.
Finiti gli elementi nell'input, se necessario, si svuota completamente la pila nell'output.[4]
Sia $\pi = \pi_L n \pi_R$, con n valore massimo in π, vale che $S(\pi) = S(\pi_L)S(\pi_R)n$

Algorithm 2 operatore S - stack sort, singola iterazione

```
1: initialize an empty stack
2: for i = 1 to n − 1 do
3:     while stack is unempty and π_i >top of the stack do
4:         pop from the stack to the output
5:     end while
6:     push(π_i)
7: end for
8: emtpy the stack in the output
```

QUEUE SORT

Per ogni elemento π_i della permutazione π in input se la coda é vuota o il suo ultimo elemento é minore di π_i, si accoda π_i, altrimenti si tolgono

elementi dalla coda ponendoli nell'output fino a che l'elemento davanti alla coda non é maggiore di π_i, poi si aggiunge π_i all'output. Si svuota la coda nell'output[5].

Algorithm 3 operatore Q - queue sort, singola iterazione

```
1: initialize an empty queue
2: for i = 1 to n − 1 do
3:     if empty queue or last in queue < π_i then
4:         enqueue(π_i)
5:     else
6:         while first in queue < π_i do
7:             dequeue(π_i)
8:         end while
9:         add π_i to the output
10:    end if
11: end for
12: emtpy the queue in the output
```

BYPASS L'operazione che pone un elemento nell'output senza passare dal contenitore si dice **bypass**. Questa viene svolta normalmente nel queuesort, ma talvolta puó essere introdotta in altri algoritmi.

OSSERVAZIONE *Bubble sort* é un caso particolare sia di *queue sort* che di *stack sort*.
Se infatti si fissa a 1 la dimensione della pila o della coda dei rispettivi operatori il comportamento che questi assumono é quello di una cella che, scorrendo l'input, contiene sempre il massimo valore trovato, mentre gli altri vengono messi nell'output.

CONTENITORI POP Un caso di studio interessante é quello in cui i contenitori di stack sort e queue sort vengano sostituiti dalla loro versione POP, cioé che quando viene eseguita un estrazione il contenitore viene svuotato completamente. Si definiscono con questa variante, gli algoritmi **pop-stacksort** e **pop-queuesort**

CLASSI DI PATTERN DI PERMTUAZIONI

[2] Siano α, β due sequenze di interi, si indica con $\alpha \subseteq \beta$ che α é una sottosequenza di β, anche se non necessariamente una sottosequenza

consecutiva.

Si dice che una permutazione δ é un pattern contenuto in una permutazione τ se esiste una sottosequenza di τ di ordine isomorfico rispetto a δ, e si indica con $\delta \preceq \tau$. Ad esempio 24153 contiene il pattern 312 perché $413 \subset 24153$.

PATTERN STANDARD Una **standardizzazione**[3] di una sequenza di numeri é un altra sequenza della stessa lunghezza in cui l'elemento minore della sequenza originale é stato sostituito da 1, il secondo minore con un 2, ecc.
Ad esempio, la standardizzazione di 5371 é 3241.

CLASSI DI PATTERN La relazione di sottopermutazione é una relazione di ordine parziale che viene studiata con dei sottoinsiemi chiamati **pattern di classi**. Ogni classe di pattern D puó essere caratterizzata dall'insieme minimo M che evita:

$$D = Av(M) = \{\beta : \mu \not\preceq \beta \forall \mu \in M\}$$

ALGORITMI E PATTERN-AVOIDANCE

É noto in letteratura [4] che una permutazione puó essere ordinata da una sola passata di stack sort se e solo se non contiene pattern 231.

PATTERN 231 Si dice che una permutazione π contiene un pattern 231 se $231 \preceq \pi$, ovvero se $\exists a < b < c : \pi = \dots b \dots c \dots a \dots$.

Allo stesso modo, sono note simile condizioni perché una permutazione possa essere ordinata da una sola passata degli altri operatoro descritti precedentemente.

Operatore	**Permutazioni ordinabili con una sola passata**
Stack sort	Av(231)
Queue sort	Av(321)
Bubble sort	Av(231,321)
Pop-stack sort	Av(231,312)
Pop-queue sort	Av(321,2413)
Pop-stack sort con bypass	Av(231,4213)

3

PROGRAMMI REALIZZATI

PERMUTASORT

Il primo programma che ho realizzato viene lanciato da linea di comando specificando come argomenti un intero n e un operatore di ordinamento X. Il suo scopo é quello di enumerare tutte le n-permutazioni ordinabili e non ordinabili con una sola passata di X, stampa anche i possibili risultati di X su n-permutazioni.

La classe `SelectorPermutations` é il core del programma, viene inizializzata passando un intero e un operatore al costruttore. Grazie alla funzione `itertools.permutations` vengono generate le n-permutazioni e , scorrendole tutte, vi si applica l'operatore X. Il risultato viene aggiunto alla lista `outcomes` e viene valutato: se é ordinato la permutazione da cui si é ottenuto viene aggiunta alla lista `sortable`, altrimenti a `unsortable`.
La classe fornisce dei *getters* per le liste e non ha altri metodi computativi, oltre al costruttore.
Dopo essere stata instanziata la classe viene usata come riferimento per i dati che ha giá calcolato nel costruttore, e non produce ulteriori risultati.
Di seguito viene mostrato il listato della classe:

```
import sys
from itertools import permutations

from src.operators import *
from src.utils import *

class selectorPermutations:
  def __init__(self,num,op):
    if num <=0:
      print("ERROR: was expecting a positive integer but got "
```

```
            +str(num))
        exit()

    # initialization of lists
    self.__sortable = []
    self.__unsortable = []
    self.__outcomes = []

    # generation of permutations
    permutations_list = list(permutations(range(1,num+1)))

    for P in permutations_list:

      # applying the operator to the permutation
      op_P_ = op(P)

      # adding outcome to the list
      if op_P_ not in self.__outcomes:
        self.__outcomes.append(op_P_)

      # adding permutations to the right list
      if isIdentityPermutation(op_P_):
        self.__sortable.append(P)
      else:
        self.__unsortable.append(P)

  def getSortable(self):
    return self.__sortable

  def getUnsortable(self):
    return self.__unsortable

  def getOutcomes(self):
    return self.__outcomes

# more lines present in the original file are omitted
# from this one due to excessive lenght since it's not of our
# interest to show further in the thesis
```

PATTFINDER

Questo programma ha lo scopo di selezionare tra tutte le permutazioni di una data dimensione quali contengono un dato pattern classico quali no.

Il programma viene lanciato da linea di comando prendendo come argomenti un intero e una sequenza di interi consecutivi eventualemente non ordinati.
L'intero rappresenta la lunghezza delle permutazioni da scorrere e la sequenza rappresenta il pattern da ricercare.
Gli argomenti vengono passati al costruttore della classe `PatternAvoid`, l'intero n viene utilizzato per generare le n-permutazioni grazie alla funzione `permutations`. Per ogni permutazione viene controllato se contiene o no il pattern e viene inserita nella rispettiva lista: `containing` o `notcontaining`.

La classe presenta i getter per le liste prodotte e due ulteriori metodi:

`patternize` serve a *standardizzare* una sequenza, si ottiene la versione ordinata della sequenza, e si sostituisce ogni valore con la posizione che occupa una volta oridnato;

`contains` verifica che una sequenza contenga un pattern o meno, grazie alla funzione `itertools.combinations` si ottengono tutte le sottosequenze (anche non consecutive), le si standardizzano e le si confrontano con il pattern da ricercare.

```
import sys
from itertools import permutations,combinations

class PatternAvoid():
  def __init__(self,n, pattern):
    if n <= 1:
      print("ERROR: expecting a whole number grater than 1 but
          got: " + str(n))
      exit()

    # checking the argument really is a classical pattern
    for i in range(1, len(pattern)):
      if i not in pattern:
        print("ERROR: expected a pattern but got: " + str(pattern))
```

```
            print("\nA pattern of lenght n must have all the numbers
                from 1 to n, " + str(i) + " not present")
            exit()

        # generating permutations
        permutations_list = list(permutations(range(1,n+1)))

        # declaring list fields
        self.__notcontaining = []
        self.__containing = []

        for p in permutations_list:
          if not self.contains(p,pattern):
            self.__notcontaining.append(p)
          else:
            self.__containing.append(p)

    # get the standard image of the number sequence:
    def patternize(self,pi):
        output = []
        sortedpi = sorted(pi)
        for p in pi:
          output.append(sortedpi.index(p)+1)
        return output

    # check if a seqence contains the given pattern
    def contains(self, seq, pattern):
        if len(seq) < len(pattern):
          return False
        subseq = combinations(seq, len(pattern))
        for s in subseq:
          if list(self.patternize(s)) == list(pattern):
            return True
        return False

    def getNotContaining(self):
        return self.__notcontaining

    def getContaining(self):
        return self.__containing
```

4

COMPOSIZIONE DI OPERATORI

[Partire dalla combinazione di S e B [1], poi spiegare l'algoritmo per S e B [3] e mostarne l'applicazione per BS, QS, QB, infine mostrare i passaggi per queuesort e mostrare BQ e SQ, facendo riferimento alla ricerca di preimmagini di pattern generici secondo Q [5]]

BIBLIOGRAFIA

[1] Michael H Albert, Mike D Atkinson, Mathilde Bouvel, Anders Claesson, and Mark Dukes. On the inverse image of pattern classes under bubble sort. *arXiv preprint arXiv:1008.5299*, 2010. (Cited on page 13.)

[2] Mathilde Bouvel, Lapo Cioni, and Luca Ferrari. Preimages under the bubblesort operator. *arXiv preprint arXiv:2204.12936*, 2022. (Cited on page 7.)

[3] Anders Claesson and Henning Ulfarsson. Sorting and preimages of pattern classes. *Discrete Mathematics & Theoretical Computer Science*, (Proceedings), 2012. (Cited on pages 8 and 13.)

[4] CONG HAN LIM. Brief introduction on stack sorting. (Cited on pages 6 and 8.)

[5] Hjalti Magnússon. Sorting operators and their preimages. *Computer Science*, 2013. (Cited on pages 7 and 13.)