Zadania domowe. Zestaw 2.2

Maciej Poleski

2 grudnia 2012

1

2

Posortować według niemalejących czasów obsługi i obsługiwać po kolei. Weźmy dowolne rozwiązanie optymalne (kolejność obsługi klientów). Zmodyfikujmy to rozwiązanie przenosząc najbardziej czasochłonnego klienta na koniec kolejki. Koszt obsługi ostatniego klienta nie zależy od kolejności obsługi - liczy się tylko czas zakończenia, który bez względu na kolejność jest taki sam (dodawanie jest przemienne). Mamy taką sytuacje (w rozwiązaniu optymalnym):

```
------X-------Y
```

`Y` jest ostatni, `X` jest najbardziej kosztowny. Jeżeli zamienimy ich miejscami, to wszystkie koszta przed `X` pozostaną takie same, koszt ostatniego klienta pozostanie taki sam, a koszta klientów od `X` do `Y` nie wzrosną ponieważ `Y`$\leqslant$`X`.

```
A[0..n-1] <- tablica z kosztami obsługi wszystkich klientów
sort(A)
result <- 0
prefix <- 0
for i <- to n):
   prefix <- prefix + A[i]
   result <- result + prefix
```

Odpowiedź jest w zmiennej `result`.

3

Zauważmy że trzecie ograniczenie jest nadmiarowe. Wszystkie elementy zbioru pustego zostały wybrane (zawsze). Więc trzecie ograniczenie jest szczególnym przypadkiem pierwszego. Pozostałe ograniczenia możemy utożsamiać z formułami logicznymi postaci

$$x_i \land x_{i+1} \land \ldots \land x_{i+k} \to x_l$$

. Wtedy warunek drugi będzie realizowany formułą dla której $x_l = \bot$ i będziemy szukać odpowiedzi na pytanie czy istnieje takie wartościowanie zmiennych $x_1 \ldots x_n$ dla którego każda formuła ma wartość 1.

Oto jak tego dokonamy: Będziemy przechowywać kolekcje wszystkich ograniczeń oraz informację pozwalającą dla zadanej zmiennej natychmiast (w czasie stałym) uzyskać formuły w których występuje. Gdy określimy że wartość pewnej zmiennej wynosi 1 - usuniemy ją ze wszystkich ograniczeń w których występuje jako poprzednik implikacji (jako element neutralny logicznej koniunkcji). Gdy zachodzi (pusto) spełnienie poprzednika implikacji wiemy że następnik musi mieć wartość 1. Jeżeli jego wartość jeszcze nie jest znana - właśnie ją poznaliśmy. Jeżeli wynosi 0 - mamy sprzeczność (rozwiązanie nie istnieje, ponieważ wszystkie zmienne których wartość określiliśmy do tej pory są określone jednoznacznie). Wykonujemy tą operację do oporu. Jeżeli udało się - wartość wszystkich do tej pory nie ustalonych zmiennych możemy uznać za 0 - dzięki temu wszystkie implikacje będą spełnione. (z fałszu wynika wszystko...).

```
queue <- kolejka zmiennych których wartość określiliśmy na 1
--przeglądnij wszystkie ograniczenia i umieść w kolejce
implikacje których poprzednik jest (pusto) spełniony--
while queue is not empty:
   x <- queue.dequeue()
   usuń wszystkie wystąpienia x ze zbioru ograniczeń
   dodaj do kolejki wszystkie zmienne które w efekcie stały
      się następnikiem implikacji z pustym poprzednikiem
   if istnieje ograniczenie którego poprzednik jest spełniony
      a następnik fałszywy:
      ROZWIĄZANIE NIE ISTNIEJE
ROZWIĄZANIE ISTNIEJE
```

A teraz jak zaimplementować wnętrze pętli, aby uzyskać oczekiwaną złożoność. Ograniczenia możemy identyfikować za pomocą identyfikatora (liczby porządkowej). Wystarczy, że dla każdego ograniczenia będziemy pamiętać ile ma poprzedników i jaki jest jego następnik (i czy jest on $\perp$). Oprócz tego dla każdej zmiennej (identyfikowanej np liczbą porządkową) zapamiętujemy w których ograniczeniach ona występuje (jako poprzednik implikacji). Dzięki temu usuwamy x w czasie proporcjonalnym do ilości jego wystąpień w ograniczeniach (zmniejszając zapamiętaną liczbą poprzedników o 1). Jeżeli jakieś ograniczenie w efekcie osiągnęło 0 poprzedników sprawdzamy jaki ma następnik i dodajemy go do kolejki (jeżeli jest on fałszywy - ROZWIĄZANIE NIE ISTNIEJE). Taki preprocesing trwa proporcjonalnie do długości formuł niemal bez względu na to jak są one wyrażone na wejściu, a po jego wykonaniu już nie korzystamy z danych wejściowych. W efekcie cały algorytm działa w czasie proporcjonalnym do długości formuł.

4

Zasadniczo jest to niemalże problem optymalnego nawiasowania. Dla każdego infiksu wyznaczymy jakie wartości może on przyjąć po nawiasowaniu (na pewno będzie co najmniej jedna wartość - jakiś wynik musi powstać).

```
T[0..n-1][0..n-1] <- T[i][j] jest zbiorem możliwych do osiągnięcia
                                wyników infiksu x_i ... x_j
T[i][i] <- {x_i}

stage1(i,j):
   if T[i][j]!=∅
       return T[i][j] -- już policzone
   for k <- i to j):
       for each x in stage1(i,k):
           for each y in stage1(k+1,j):
               T[i][j] <- T[i][j] ∪ {x∘y}

stage2(i,j,z):
   if i = j:
       print z
       return
   print '('
   for k <- i to j):
       for each x in stage1(i,k):
           for each y in stage1(k+1,j):
               if x∘y = z:
                   stage2(i,k,x)
                   print ')∘('
                   stage2(k+1,j,y)
                   break wszystkie pętle

   print ')'

main:
   stage1(0,n-1)
   if a ∈ T[0][n-1]:
       stage2(0,n-1,a)
   else:
       NIE DA SIĘ
```

Uruchomić `main`. Złożoność jest taka sama jak w optymalnym nawiasowaniu (wykład).