IV003, Fakulta Informatiky, Masarykova Univerzita

Homework 2

Jakub Senko, Štefan Uherčík

15. apríla 2014

Príklad 1

Zaveďme všeobecnú reprezentáciu budov. Každá uvažovaná budova sa dá reprezentovať ako množina dvojíc (x_k, h_k), určujúcich výšku budovy h na súradnici x. Zápis sa dá zjednodušiť usporiadaním bodov vzostupne podľa x. Stačí uvažovať len tie dvojice, ktoré označujú miesto, v ktorom nastáva zmena výšky budovy. Tento zápis je ekvivalentný so zápisom použitým v zadaní

$$(1, \mathbf{5}, 5) \sim ((1, 5), (5, 0)) \tag{0.1}$$

ide len o vnútornú reprezentáciu za účelom zjednodušenia algoritmu.

Merge

Uvažujme algoritmus MERGE, ktorý z reprezentácie dvoch budov vypočíta reprezentáciu ich siluety.

Algoritmus využíva object BUILDING_ITERATOR pomocou ktorého je možné postupne prechádzat reprezentáciou danej budovy. Obsahuje tri metódy.
NEXT_COORDINATE_EXISTS a NEXT_COORDINATE_POSITION sú triviálne a neposúvajú pozíciu iterátora. Tretia metóda, GET_HEIGHT(x) vráti výšku budovy na zadanej súradnici. Táto metóda spôsobí dostatočný posun iterátora v prípade, že zadaná pozícia je väčšia alebo rovná ako NEXT_COORDINATE_POSITION. Kedže iterátor

je jednorázový, túto metódu je nie je možné zavolať s argumentom menším ako v predchádzajúcom volaní. Iterátor si jednoducho pamätá poslednú výšku.

Samotný MERGE pracuje s dvoma iterátormi, pre každú budovu jeden a výstup postupne ukladá do samostatného zoznamu. Základom je *while* smyčka, ktorá sa vykoná ak aspoň pre jeden s iterátorov platí NEXT_COORDINATE_EXISTS. Algoritmus potom vybere menšie x z NEXT_COORDINATE_POSITION a zavolá metódu GET_HEIGHT na oboch iterátoroch. Následne vybere väčšiu z výšok, h a zavolá funkciu TRY_ADD, ktorá jednoducho vloží novú súradnicu (x, h) do výsledného zoznamu v prípade, že sa výška siluety zmenila (čo nemusí nastať).

Tento algoritmus funguje pre ľubovolné reprezentácie s dĺžkou n_1, n_2 v čase $\mathcal{O}(n_1 + n_2)$, čo je $\mathcal{O}(n)$ pre budovy s rovnako veľkou reprezentáciou. Zdôvodnenie je jednoduché - využíva jednosmerný iterátor na jedno použitie pre každú reprezentáciu - a teda každú súradnicu spracuje práve raz. Algoritmus je konečný pretože pri každom priechode cyklom metóda GET_HEIGHT posunie aspoň jeden z iterátorov.

Rozdeľ a panuj

Výslednú siluetu dosiahneme aplikovaním funkcie MERGE na vhodné podproblémy. Toto delenie funguje rovnako ako pri algoritme *merge sort*. Funkcia COMPUTE_SILHOUETTE zoberie ako argument množinu reprezentácii budov. Ak táto množina obsahuje jednu budovu, tak ju vráti. Ak dve budovy, zavolá na nich MERGE a vráti výsledok. Ak viac, rozdelí množinu na dve rovnaké (s rozdielom jednej budovy v prípade nepárneho počtu) množiny, rekurzívne sa na oboch zavolá a výsledok znovu spojí pomocou MERGE a vráti. Týmto spôsobom funkcia COMPUTE_SILHOUETTE vždy vráti merge všetkých spojich argumentov (merge nezávisí na poradí).

Zložitosť závisí na počte MERGE operácii a veľkosti ich vstupu. Na každej úrovni rekurzie je suma veľkosti všetkých reprezentácií rovnaká (n dĺžky 2 na začiatku vs dve dlhé n na konci, kde n je počet budov) a počet úrovní je $\log_2 n$. Výsledná zložitosť je teda $\mathcal{O}(n \log n)$

Príklad 2

Algoritmus vykoná všetky možné rozdelenia vstupného reťazca na 2 časti, prefix a suffix. Ak je daný reťazec veta, existuje aspoň jedno rozdelenie také, že suffix je slovo a prefix je znovu veta. Tento test sa vykoná v tele cyklu, kde sa rekurzívne použije IS_SENTENCE na danom prefixe vstupného slova. Premenná *result* je true práve vtedy ak je aspoň jeden z týchto testov je true.

```
function IS_SENTENCE(w[1...n])
    result = IN_DICTIONARY(w[1...n]);
    for i = 1...n do
        item = IS_SENTENCE(w[1...i]) ∧ IN_DICTIONARY(w[i + 1...n]);
        result = result ∨ item;
    end for
    return result;
end function
```

Pri každom z rekurzívnych volaní je argument funkcie IS_SENTENCE *vždy prefixom* argumentu volajúcej funkcie. Pri tomto rekurzívnom algoritme môže existovať v strome rekurzie viacero ciest k volaniu IS_SENTENCE s rovnakým argumentom, avšak každý reťazec má konečné množstvo prefixov, konkrétne $n - 1$. Preto stačí vykonať iba lineárne množstvo volaní. Tieto volania sa dajú *usporiadať* tak, že sa funkcia postupné volá pre rastúci prefix. Táto myšlienka je základom nasledujúcej dynamickej varianty funckie IS_SENTENCE.

Predstavme si nasledovný prípad Na vstup dostane algoritmus reťazec o dĺžke n. Pri overovaní jednotlivých rozdelení nájde slovo o dĺžke a (a zároveň sa rekurzívne zanorí na prefixe o dĺžke (n-a)) a ďalej pokračuje v overovaní ďalších rozdelení (s prefixami dĺžky (n-a+1),(n-a+2),...).
Algoritmus sa rekurzívne zavolá na reťazci o dĺžke n-a. Pri týchto volaniach však overuje prefixy s dĺžkami 1 .. n-a. Tieto prefixy však overoval aj predošlý priechod algoritmu.
Môžeme povedať, že volanie funkcie IS_SENTENCE aplikované na reťazci dĺžky n je závislé na všetkých možných volaniach funkcie IS_SENTENCE aplikovaných na prefixoch tohto reťazca.
Z tohoto dôvodu je výhodné, ak vypočítame IS_SENTENCE na prefixoch pôvodného reťazca a výsledky týchto volaní si uložíme do rovnomenného asociatívneho poľa. Kľúč tohto poľa bude tvorený prefixom pôvodného reťazca, hodnota bude typu boolean.

```
function IS_SENTENCE(w[1...n])
    for i = 1...n do
        result = IN_DICTIONARY(w[1...i]);
        for j = 1...i − 1 do
            item = SENTENCE_DATA(w[1...j]) ∧ IN_DICTIONARY(w[j+1...i−1]);
```

```
            result = result ∨ item;
        end for
    end for
    return result;
end function
```

Korektnosť:
Konvergencia: Prvy cyklus sa vykoná pre každý prefix slova w. V cykle sa hodnota premennej i reprezentujucej dlžku prefixu pri každom priechode zvýši o 1 a iterovanie skončí, keď premenná i dosiahne hodnotu n.
Vnorený cyklus sa vykoná pre každý prefix tohto prefixu.
Jediné miesta, u ktorých hrozí, že algoritmus nezastaví, sú volania cyklov.
V cykle s iterujúcou premennou j sa zaručene pri každom priechode zvýši hodnota premennej j a iterovanie skončí, keď premenná j dosiahne hodnotu i-1.
Parciálna korektnosť: Dôkaz, že algoritmus vráti true pri validnej postupnosti slov pomocou matematickej indukcie: 1.) S0 = w0 je veta ktorá sa skladá z jedného slova w0 (w0 sa nachádza v slovníku). Pre takúto vetu vráti algoritmus true. Dôvod: položky subresults pre prvok IS_SENTENCE(w0) budú obsahovať položku dict(w0), ktorá sa vyhodnotí na true. Na položky je aplikovaná funkcia logiký súčet a pretože obsahujú minimálne jednu položku z hodnotou true, algoritmus vráti true.
2.) Predpokladáme, že pre vetu s1 zloženú z k slov: s1 = w1.w2...wk vráti korektnú odpoveď true. Predpokldáme, že pre vetu s2 zloženú s2 = s1.w(k +1) algoritmus taktiež vráti true. Jeden z prefixov vety s2 musí byť reťazec zložený zo slov w1...wk. Keďže algoritmus prechádza všetky prefixy, nastane situácia, jedna z položiek v poli subresults pre prvok s2 bude IS_SENTENCE(s1) logical_and dict(w(k+1)).
Dôkaz, že algoritmus vráti true pri nevalidnej postupnosti slov
Zložitosť: Cyklus s iterujúcou premennou i bude vykonaný n krát.
Vnorený cyklus s iterujúcou premennou j bude vykonaný (i-1) krát, a plaží, že (i-1) < n. Všetky operácie použité v ňom majú konštantnú zložitosť. Volanie apply logical_or on subresults bude mať rovnakú zložitosť ako vnorený cyklus (vzhľadom k tomu, že počet jeho položiek zodpovedá počtu iterácií).
$\mathcal{O}(n^2)$

Príklad 3

Vstupom algoritmu bude: pole pravdepodobností: $C[p(1),..,p(n)]$
k - počet padnutých orlov
PVD - pravdepodobnostná funkcia
n Problém je možné definovať nasledovne: vypočíť pravdepodobnosť, že v poli padne k orlov z n mincí
táto pravdepodobnosť je ekvivaletná súčtu pravdepodobností nasledovných prípadov:
1.) pravdepodobnosť prípadu, že posledná minca bude orol táto pravdepodobnosť je ekvivalentná súčinu čísla p(n) a pravdepodobnosti, že medzi prvými n-1 mincami bude k-1 orlov
2.) pravdepodobnosť prípadu, že posledná minca nebude orol táto pravdepodobnosť je ekvivalentná súčinu čísla (1-p(n)) a pravdepodobnosti, že medzi prvými n-1 mincami bude k orlov
Tento poznatok nám umožňuje definovať jednoduchý rekurzívny algoritmus (v ktorm zároveň ošetrujeme krajné prípady n=k a n=0)

```
function PVD(C[p1,..,pn],k)
    if k=0 then
        return PVD(C[p(1),..,p(n-1)],0)*(1-p(n));
    end if
    if k=n then
        return PVD(C[p(1),..,p(n-1)],k-1)*p(n);
    end if
    return PVD(C[p(1),..,p(n-1)],k-1)*p(n) + PVD(C[p(1),..,p(n-1)],k)*(1-p(n));
end function
```

Ak rozpíšeme vetvenie algoritmu vykonávanie bude vyzerať približne nasledovne PVD(C[p(1),..,p(n)],k) = PVD(C[p(1),..,p(n-1)],k-1)*p(n) + PVD(C[p(1),..,p(n-1)],k)*(1-p(n))
PVD(C[p(1),..,p(n-1)],k) = PVD(C[p(1),..,p(n-1)],k-1)*p(n-1) + PVD(C[p(1),..,p(n-1)],k)*(1-p(n-1))
PVD(C[p(1),..,p(n-1)],k-1) = PVD(C[p(1),..,p(n-1)],k-2)*p(n-1) + PVD(C[p(1),..,p(n-1)],k-1)*(1-p(n-1))
... Z predchádzajúceho zápisu volaní funkcií je možné vidieť, že PVD(C[p(1),..,p(n-1)],k-1) sa zavolá 2 krát na jednej úrovni rekurzívneho stromu. Využijeme techniku dynamického programovania, aby sme sa vyhli opakovanému volaniu funkcie PVD na rovnakých parametroch. Z algoritmu je zreteľné, že volanie funkcie PVD, ktorá berie ako parameter pole o dĺžke a, je závislá výlučne na volaniach funkcií PVD, ktoré berú ako parameter pole o dĺžke a-1. Z tohoto dôvodu je výhodné, ak vypočítame najprv. funkcie s parametrami PVD(C[p(1)],0), PVD(C[p(1)],1), PVD(C[p(1),p(2)],0),... ,ich výsledky si budem ukladať do asociatívneho poľa a postupným volaním sa dopracujem k hodnote PVD(C[p(1),..,p(n)],k), ktorá je výsledkom celého problému. Pre tento účel vytvoríme asociatívne pole s názvom PVD, v ktorom kľúče budú mať tvar: (C[p(1),..,p(n)],k) a hod-

noty budú obsahovať napočítanú pravdepodobnosť. Na naplnenie tohto poľa vytvoríme jednoduchú nerekurzívnu funkciu:

```
function countPVD(C[p1,..,pn],k)
    PVD([p(1)],0) = p1
    PVD([p(1)],1) = (1-p1);
    for i = 1 .. n do
        bottom = max(0,k-(n-i));
        up = min(i,k);
        for j = bottom .. up do
            if k=0 then
                PVD(C[p(1),..,p(i-1)],0)*(1-p(i));
            else
                if k=n then
                    PVD(C[p(1),..,p(i-1)],j-1)*p(i);
                else
                    PVD(C[p(1),..,p(i-1)],j-1)*p(i) + PVD(C[p(1),..,p(i-1)],j)*(1-p(i));
                end if
            end if
        end for
    end for
    return PVD(C[p(1),..,p(n-1)],k);
end function
```

Zložitosť: Cyklus s iterujúcou premennou i sa vykoná n krát. V ňom sa vnorený cyklus iterujúcou premennou j vykoná vždy (up - bottom) krát. V každom cykle bude hodnota premennej bottom minimálne 0 a hodnota premennej up maximálne k, z čoho vyplýva, že počet týchto cyklov bude maximálne k. Je zaručené, že k < n a teda zložitosť celého algoritmu bude patriť do triedy $\mathcal{O}(n^2)$

Príklad 4

rekurzívny algoritmus
Cenu optimálneho rozdelenie áut do autosalónov je možné vypočítať pomocou jednoduchého rekurzívneho algoritmu. Číslo auta je zároveň jeho index riadku v matici cien. Parameter n v algoritme bestPrice udáva počet áut ktoré ešte neboli priradené do autosalónu a kedže sa autá priradujú postupne, tak udáva zároveň aj číslo nasleujúceho nepriradeného auta. Parameter freePlaces udáva počet neobsadených miest v jednotlivých autosalónoch (číslo na indexe i v tomto poli udáva počet volných miest v autosalóne i). V každom rekurzívnom volaní priradíme auto do každého autosalónu az týchto priradení vrátime maximálnu cenu. Rekurzia sa zastaví v prípade že sme priradili už všetky autá.

```
function BEST_PRICE(n,freePlaces[])
    if n == 0 then
        return 0
    end if
    values = [] of number
    for i = 1 .. freePlaces.size do
        if freePlaces[i] != 0 then
            freePlacesCopy = copy of freePlaces;
            freePlacesCopy[i] = freePlacesCopy[i] - 1;
            values[i] = best_price(n-1,freePlacesCopy) + C[n][i];
        end if
    end for
    return max(values);
end function
```

Technika dynamického programovania

Každé volanie funkcie bestPrice na matici s počtom riadkov k potrebuje výsledky volaní funkcie betPrice na matici s počtom riadkov k-1. Toto nám jasne definuje závislosť a teda možné usporiadanie volaní.
Vytvorím štruktúru bestPriceData, ktorá bude typu asociatívne pole, kľúče bude dvojica parametrov predávaná funkcie bestPrice. Touto štruktúrov memoizujeme volanie bestPrice.
Aby sme zaznamenali všetky možné kombinácie parametrov n a freePlaces, vo funkcii bestPrice sme vytvorili funkciu findSpecialPermutations. Táto funkcia vráti všetky možnosti ako môže vyzeraž pole freePlaces pre sum áut v prípade že do jedného autosalónu môžeme daž maximálne threshold áut.

```
function FINDSPECIALPERMUTATIONS(sum,threshold)
    if sum = 0 then
        return [0,0,0]
    end if
    permutations = [] of [];
    bottom1 = max(0,sum - 2*threshold);
```

```
    upper1 = min(sum,threshold);
    for  i = bottom1 .. upper1  do
        permutation = [] of length 3
        permutation[1] = i;
        bottom2 = max(0,sum - i - threshold);
        upper2 = min(sum - i,threshold);
        for j=bottom2 .. upper2 do
            permutation[2] = j;
            permutation[3] = sum - (i+j);
        end for
      permutations.add(permutation);
    end for
    return permutations;
end function
```

Nasledovná nerekurzívna funkcia postupne priradí do štruktúry bestPriceData hodnoty najlepších možných cien pre danú konfiguráciu.
Pri každom prechode vonkajším cyklom využívame hodnoty v bestPriceData vypočítané v predchádzajúcich priechodoch cyklom.

```
function countBestPrice(C) bestPriceData[(0,[0,0,0])] = 0;
    for i = 1 .. C.rows do
        permutations = findSpecialPermutations(i,C.rows/3);
        for permutation in permutations do
            items = [] of number;
            for  j = 1 .. permutation.size do
                modifiedPermutation = copy of permutation
                if modifiedPermutation[j]>0 then
                    modifiedPermutation[j] = modifiedPermutation[j] - 1;
                    items.add(bestPrice(i-1,modifiedPermutation) + C[i][j]);
                end if
            end for
            bestPrice[(i,permutation)] = max(items);
        end for
    end for
    return bestPrice with key(C.rows);
end function
```

Pre krátkost tento algoritmus počíta len výšku najlepšej ceny za akú je možné autá predať, ale je triviálne modfikovateľný aby si spolu s cenou pametal aj konkretne priradenie aut do autosalonov. Namiesto ceny je stačí uložiť do bestPriceData dvojicu (cena, priradenie). Priradenie je pole o dlžke C.rows a hodnota na indexe i v tomto poli vyjadruje číslo autosalónu do ktorého bude auto priradené, prípadne 0 ak eťe nie je priradené.
Nasledujúci algoritmus toto implementuje:

combinations = nájdeme všetky kombinácie práve a císel z N0, ktoré sú menšie ako C.rows a, ktoré dávajú súčet i;

```
function countBestPrice(C) bestPriceData[(0,[0,0,0])] = 0;
    for i = 1 .. C.rows do
        permutations = findSpecialPermutations(i,C.rows/3);
        for permutation in permutations do
            items = [] of item (price -> number, distribution -> [] of number);
            for  j = 1 .. permutation.size do
                modifiedPermutation = copy of permutation
                if modifiedPermutation[j]>0 then
                    modifiedPermutation[j] = modifiedPermutation[j] - 1;
                    item it;
                    aPrice = bestPrice(i-1,modifiedPermutation);
                    it.price = aPrice.price + C[i][j];
                    it.distribution = aPrice.distribution;
                    it.distribution[i] = j;
                    items.add(it);
                end if
            end for
            highestPriceIndex = index into items for item with highest price;
            bestPrice[(i,permutation)] = items[highestPriceIndex];
        end for
    end for
    return bestPrice with key(C.rows);
end function
```

Algoritmus je rozšíriteľný pre väčší počet autosalónov, je však potrebné zmeniť implementáciu funkcie findSpecialPermutations.

Konvergencia

Parciálna korektnosť

Príklad 5

Hladový algoritmus nenájde správne riešenie pre druhý a tretí problém.
Protipríklad:
Dokument s dĺžkou riadku: 12
Zoznam slov obsahuje slová s dĺžkami 5,4,4,12

Algoritmus umiestni slová nasledovne:
5,4
4
12

Druhý slovný problém:
Celková penalizácia bude $3^2 + 8^2 + 0 = 9 + 64 = 73$

Optimálne riešenie je však:
5
4,4
12
pri ktorom bude celková penalizácia $7^2 + 4^2 = 49 + 16 = 64$

Tretí slovný problém:
Celková penalizácia bude: max(3,8,0) = 8

Optimálne riešenie je však:
5
4,4
12
pri ktorom bude celková penalizácia max(7,4,0) = 7