Norsk Mesterskap i Programmering inom ACM ICPC
Svenskt Mästerskap i Programmering inom ACM ICPC

Solution Ideas 2004

PIZZA

Kökets position definieras av ett val av en vertikal och en horisontell gata. Dessa gator kan bestämmas oberoende av varandra, vilket blir uppenbart om man betraktar kostnadsfunktionen lite närmare. Vi får alltså två endimensionella optimeringsproblem. I dessa problem är vikten för en gata antalet leveranser som gjorts vid någon korsning längs gatan.

Betrakta valet av horisontell gata och antag att vi har en kandidat, G. Om vi istället väljer nästa gata norrut, kommer varje leverans till gator norr om G att få sin kostnad minskad med ett, medan leveranser till G och gator söder om G får sina kostnader ökade med ett. En konsekvens av detta är att den optimala gatan måste ha jämnast möjliga fördelning mellan antalet leveranser norr och söder om sig. Mer formellt: om vi betraktar alla enskilda leveranser och sorterar dem i gatuordning, skall mittleveransen i den sorterade sekvensen vara till G (vid ett jämnt antal gator kan det finnas två mediangator som båda ger samma optimala resultat).

Ball Bearings

Låt n vara antalet kulor i kullagret. Vi kan anta att de ligger jämnt fördelade längs ringen. Centrumpunkterna för kulorna bildar hörnen i en regelbunden n-hörning (nu borde man egentligen ha en figur).

n-hörningenens hörn ligger på en cirkel med radien $R = (D - d)/2$ och dess sidor har längd $L = 2 * R * sin(pi/n)$, och L ska vara $>= s + d$

Eftersom n ska vara så stort som möligt får vi

$n = floor(PI/asin((s + d)/(D - d)))$

Testdata är valt så att små avrundningsfel inte ska ställa till problem.

NIM

Anta att vi har $S = 2, 5, 6, 9$

Högar av storlek 0 och 1 beter sigsom nim-högar av storlek 0, för vi har inga giltiga drag från dem. Vi säger att högar av storlek 0 och 1 har nim-värde 0.

Högar av storlek 2 och 3 har nim-värde 1, för vi har bara drag till (högar som beter sigsom) högar av storlek 0, dvs högar med nim-värde 0.

Från högar av storlek 4 kan vi bara flytta till högar med nim-närde 1. Om de skulle ha nim-värde 2borde vi kunna flytta även till högar med nim-värde

0. Det visar sig att det sökta nim-värdet är 0.

Vi kan se det som att högar av storlek 4 beter sigsom Nim-högar av storlek 0, men att vi dessutom kan flytta till högar med nim-värde 1. Denna extra möjlighet kan dock aldrig vara det enda vinnande draget efterssom motståndaren alltid kan flytta tillbaka till en hög med nim-värde 0 från en hög med nim-värde 1.

Vi fortsätter på samma sätt att räkna ut nim-värdet för alla högstorlekar (upp till övre gränsen 10000). Låt oss kalla nim-värdet för en hög av storlek i för $nim[i]$. När vi ska beräkna $nim[i]$ tittar vi på $nim[i-S[j]]$ för $j = 1...|S|$. Det minsta icke-negativa tal som inte finns bland dessa är $nim[i]$

För $S = 2, 5, 6, 9$ får vi slutligen:
Högstorlek: 0 1 2 3 4 5 6 7 8 9 10
Nim-värde: 0 0 1 1 0 2 1 3 0 2 1 ... och sedan upprepas samma följd.

Låt oss nu anta att vi vill avgöra om en position med högar av storlek 2, 4, 5 och 9 är en vinnande position.

Högstorlek: 2 4 5 9
Nim-värde: 1 0 2 2 (från tabellen ovan)
Xor-summa: 1 xor 0 xor 2 xor 2 = 1

Detta är en alltså en vinnande position.

(Dessutom kan vi se att ett vinnande drag måste vara ett drag som ger första högen nim-värde 0, andra högen nim-värde 1, eller någon av de två sista högarna nim-värde 3.

Detta ger de vinnande dragen:

2 -> 0, 2 -> 1, 4 -> 2, 4 -> 3, 9 -> 7)

Sylvester

Givet en någorlunda snabb funktion som beräknar värdet för en given koordinat är problemet i princip löst. Denna funktion löses lämpligtvis med rekursion genom att utnyttja den rekursiva definitionen av matrisen. Man kontrollerar vilken kvadrant koordinaten ligger i och reducerar problemet till motsvarande koordinat i den kvadraten (och därmed halveras problemet). Om kvadranten är den nere till höger ska resultatet från det rekursiva anropet förstås negeras.

Amiga

Ett backtrackingproblem.

Vi har fem olika kategorier: färg, datortyp, nationalitet, namn och programspråk som ska fördelas på fem olika rum. Detta ger $(5!)^5 > 10^{10}$. Dvs på tok för mycket för att testa alla tänkbara kombinationer. Det gäller därför att

börja med att lägga ut någon kategori, exempelvis datortyp, och förutsätta att en viss permutation av datorer råkar gälla, se om detta står i konflikt med någon regel, och därefter fortsätta med att lägga ut någon annan typ ifall det inte finns några konflikter etc.

Varefter man börjar förutsätta att en permutation gäller bör man också förutsätta att alla direkta konsekvenser av detta gäller.

Har man exempelvis:

anna left-of danish
och
danish left-of red

känner man ju till rummet som anna resp. danish tillhör givet att man vet red's rumsnummer. Detta kan göras mha bredden-först-sökning. När man gör ett antagande ang reds rumsnummer lägger man till alla fakta som följer av detta antagande i en kö. Dessa antaganden processar man därefter i tur och ordning och fyller efter hand på med konsekvenserna av dessa etc.

Notera även att danish left-of red"är ekvivalent med red right-of danish". finnish same-as pascal"är såklart ekvivalent med pascal same-as finnish"etc.

Detta minskar både antalet kombinationer att prova, och gör dessutom att vi inte behöver slutföra den backtracking-grenen eftersom vi redan fått fram den informationen vi önskar. Vi måste dock fortsätta att testa andra kombinationer för att se om det kan finnas andra tänkbara amiga-ägare.

Om man hittat en lösning som inte bryter mot någon regel bör man spara undan vem som äger Amigan i denna. Hittar man en lösning till som inte bryter mot regler, men som har en annan Amiga-ägare vet vi att det finns mer än en tänkbar ägare till Amigan, och vi kan avbryta.

Alternative solution

We define a recursive function that assigns a value to a specific property for a specific house. For instance, *Which color is house 4?*. We try all property values that has not already been used. For each such assignment, check that it doesn't violate any facts (only the facts that are about this specific property needs to be checked). If valid, do a recursive call. If not, try next property value.

This solution would be immensly slow if the choice of property and house was arbitary. Selecting which property and house is the trick that makes this solution work: We pick the property/house combination which has the least number of legal property values (after verifying with the facts). To determine this, at the start of the recursive function, try all combinations of properties and houses, and count the number of legal values (this will require many loops and checks with the facts; however, the benefits are enormous). For instance, we immediately find out if we've reached a contradiction, by realizing that,

say, the property name for house 2 can't be assigned to and value without violating the facts.

There remains one optimization needed: When a solution is found (all property values have been assigned), save the current Amiga owner. Now restart the process, but with the extra condition that the Amiga owner must be somebody else than what was previously assigned. If we now still find a solution, then obviously we cannot identify the Amiga user.

Lazy Evaluation

The main challenge in this task was to understand what is to be done, and then to parse the input properly and choose a good representation so that the evaluation can be done in a smooth way.

In order to be able to evaluate the expressions, we need to remember all function definitions, and be able to replace the parameters by arguments in the function bodies. For lazy evaluation, it can be advantageous to keep only one copy of the argument even if it is referred to from several places in the function body, so that once it gets evaluated, all occurences will contain the evaluated value.

A closer look at the expressive power of the language will disclose that each expression evaluates directly to some function (built-in or user-defined) - so it can be represented as name, or to a number. A function evaluation cannot be result of evaluation of an expression. In order to be able to lookup the function bodies, we can sort the functions based on their names alphabetically using the quick sort provided by standard API.

The rest is just a mechanical manipulation with structures - passing through the tree of function bodies, copying and replacing the arguments, and keeping accounting of the use of the built-in operators.

Holidays

The algorithm is trying to put the start of the journey into each of the possible places. For each place it will investigate all possible top stations that can be reached from this place and then find the best possible ratio that can be achieved by reaching the top station from the bottom by lifts and then returning back using slopes.

To speed up this search, the algorithm topologically sorts the places (the sort is valid for all bottom and top stations). And for each starting location, it determines the shortest paths to all other reachable stations using lifts, and the longest paths from all stations from where it can be reached by slopes. The best ratio for a pair of a bottom and a top station that can be obtained is then the ratio of the (time) lengths of the shortest and the longest path determined for this pair. From all the best ratios for different pairs the algorithm chooses the globally best ratio. In order to determine the

journey after finding the best ratio, we also keep the predecessor tree that is constructed when searching for the shortest and the longest paths.

Tourist

Consider the following easier version of the problem: Find the way from 1,1 to w,h (walking right or down) so we get the most number of *. This a fairly easy dynamic programming problem, with the following recurrence relation:

```
           # :  -inf
sq[x,y] = . :  max(sq[x-1,y],sq[x,y-1])
           * :  max(sq[x-1,y],sq[x,y-1]) + 1
```

In the original problem, applying this algorithm twice (and removing the * after the first pass) doesn't work, which is shown by the second sample input (this is a greedy algorithm that almost always gives an answer very close to the optimal solution).

The key to problem is to realize that instead of walking to the SE corner and then back to the NW, we can walk twice to the SE (this is the same thing), and do these walks simultaneously. When done simultaneously, we can easily handle the case where the two paths cross, and make sure to only count the * once in those cases.

What we end up with is a dynamic programming problem similar to the one above, with one extra parameter. We need three loops - one for time and two for the x-coordinate of the two paths (the y-coordinate simply follows from the formula x-coordinate + y-coordinate = time). The recurrence then become much like the one above, although slightly more complicated since we have two paths instead of one, and we have to do max over four values instead of two: (formul below ignores #)

```
sq[time,x1,x2]=max(sq[time-1,x1,x2],sq[time-1,x1-1,x2],
                   sq[time-1,x1,x2-1],sq[time-1,x1-1,x2-1]) +
                   number of * at square x1,y1 and square x2,y2
                   (only counting * once if x1,y1==x2,y2)
```