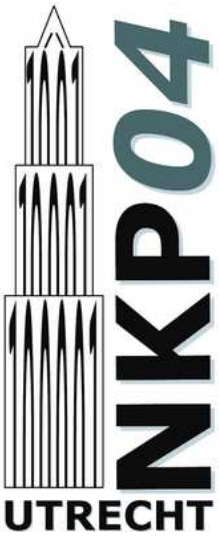
Uitwerkingen problemen

Nederlands Kampioenschap Programmeren 2004

23 oktober 2004



A Bustijden

Probleem

Gegeven een beschrijving van de vertrektijden en onzekerheden daarin van bussen op een gegeven route, moeten we het vroegst mogelijke en het laatst mogelijke tijdstip van aankomst uitrekenen.

Oplossing

Merk op dat het voldoende is om bij elke opeenvolgende halte uit te rekenen wat de vroegst en laatst mogelijke tijdstippen van aankomst zijn, noem deze `vroeg`[i] en `laat`[i]. Bij de eerste halte ($i = 0$), geldt `vroeg`[0] = "09:00" = `laat`[0].

Stel nu dat bus hh:mm (a) van halte i naar $i + 1$ rijdt. We kunnen vroeg[i+1] berekenen uit vroeg[i] met behulp van de volgende observaties:

- hh:mm + a < vroeg[i]: Deze bus mis je sowieso
- hh:mm − a > vroeg[i]: Bus komt om hh:mm − a
- anders: Bus komt om `vroeg`[i]

`vroeg`[i+1] is dan het minimum over alle bussen van i naar $i+1$ van de mogelijke aankomsttijden.

Om laat[i+1] uit de rekenen doen we iets soortgelijks:

- hh:mm − a < `laat`[i]: Deze bus mis je sowieso
- anders: Bus komt om hh:mm + a

`laat`[i+1] is dan het minimum van alle mogelijke aankomsttijden.

Bij een implementatie is het handig om alle tijden in minuten te rekenen, dus "09:00"= 540, "23:59"= 1439, etc. In C++ krijgen we dus zoiets als (met N het aantal bushaltes)

```
vroeg[0] = 540;
laat[0] = 540;
for (int i=0; i<N; i++) {
    int newvroeg=99999, newlaat=99999;
    int n,dt;
    cin >> n >> dt;

    for (int j=0; j<n; j++) {
        int h,m,t,a; char dum;
        cin >>h>>dum>>m>>dum>>a>>dum;
        t=60*h+m;
        if (t-a >= laat[i]) newlaat = min(newlaat, t+a+dt);
        if (t+a >= vroeg[i]) newvroeg = min( newvroeg, max(t-a,best)+dt );
    }

    vroeg[i+1] = newvroeg;
    laat[i+1] = newlaat;
}
```

Het is duidelijk dat deze code $O(n)$ tijd vergt per etappe. De totale tijd is dan $O(n_1 + \ldots + n_N)$, dus lineair in het totale aantal bussen.

Let er bij de uitvoer op dat de tijden in het correcte formaat gegeven worden, dus "09:15" en niet "9:15".

B Mikado

Probleem

Dit probleem is op te splitsen in twee deelproblemen:

- Het geometrische probleem om te bepalen of twee stokjes elkaar raken.
- Het graafprobleem van het bepalen van verschillende verbonden deelgrafen. Hierbij moeten we ieder stokje zien als een punt in de graaf en twee stokjes (punten) zijn verbonden als de stokjes elkaar raken.

Als je wat ervaring hebt met geometrische problemen, dan weet je waarschijnlijk dat het eerste deelprobleem inzichtelijk erg simpel is, maar knap lastig kan zijn om in de praktijk te programmeren.

Oplossing

De geometrie

Om dit op te lossen, hoeft er maar een functie geschreven te worden die test of twee lijnstukken elkaar raken. Hierbij komen echter wel een aantal vervelende details kijken.

Om dit aan te pakken, is het het handigst om een nieuw datatype te maken om 2D punten en lijnen te representeren (met vectoren). Hierop kunnen dan geometrische operaties gedefinieerd worden, zoals optellen, aftrekken, inproduct en uitproduct.

Dan moeten er twee gevallen onderscheiden worden: of de lijnstukken parallel lopen of niet. Dit is het geval als het uitproduct tussen de twee richtingsvectoren (eind- min beginpunt) nul is.

Als ze niet parallel lopen, dan snijden de lijnstukken, als de eindpunten van ieder lijnstuk niet aan dezelfde kant van het andere lijnstuk liggen en anders snijden ze niet.

Het geval dat de lijnstukken parallel lopen, is qua implementatie lastiger. Als de lijnen niet in elkaars verlengde liggen, snijden ze sowieso niet. Indien wel, dan moet gekeken worden of ze elkaar overlappen of raken.

Wiskundige pseudo-code hiervoor is als volgt, waarbij $\vec{p_1}, \vec{p_2}, \vec{q_1}, \vec{q_2}$ de eindpunten van de lijnstukken p en q zijn:

FUNCTION intersect(p, q)
  $\vec{\Delta p} = \vec{p_2} - \vec{p_1}$
  $\vec{\Delta q} = \vec{q_2} - \vec{q_1}$
  IF $\vec{\Delta p} \times \vec{\Delta q} = 0$ THEN // parallel
    IF $\vec{\Delta p} \times (\vec{q_1} - \vec{p_1})$ THEN RETURN false
    IF $(\vec{q_1} - \vec{p_1}) \cdot (\vec{q_1} - \vec{p_2}) \leq 0$ OR $(\vec{q_2} - \vec{p_1}) \cdot (\vec{q_2} - \vec{p_2}) \leq 0$ OR
      $(\vec{p_1} - \vec{q_1}) \cdot (\vec{p_1} - \vec{q_2}) \leq 0$ OR $(\vec{p_2} - \vec{q_1}) \cdot (\vec{p_2} - \vec{q_2}) \leq 0$ THEN RETURN true
    RETURN false
  ELSE // niet parallel
    IF $\left(\vec{\Delta q} \times (\vec{p_1} - \vec{q_1})\right) \cdot \left(\vec{\Delta q} \times (\vec{p_2} - \vec{q_1})\right) > 0$ THEN RETURN false
    IF $\left(\vec{\Delta p} \times (\vec{q_1} - \vec{p_1})\right) \cdot \left(\vec{\Delta p} \times (\vec{q_2} - \vec{p_1})\right) > 0$ THEN RETURN false
    RETURN false
  END IF
END FUNCTION

De graaf

Het tweede deelprobleem is op te lossen met een "floodfill". We kiezen een willekeurig punt (stokje) in de graaf, dat we nog niet gehad hebben en gaan vanaf daar alle buren langs, totdat we geen nieuwe punten meer vinden. Het aantal bezochte punten is de grootte van die stapel. Dit herhalen we totdat alle stokjes doorlopen zijn.

Hierna is het een kwestie van de bijgehouden data van het aantal stapels van de verschillende groottes in het goede formaat uit te voeren.

C Avondvierdaagse

Probleem

Bondig geformuleerd is het probleem: gegeven een gesorteerde rij van n intervallen $[b_i, e_i]$ die niet overlappen, overdek deze met een minimum aantal intervallen van lengte l.

Oplossing

De oplossing van dit probleem wordt gevonden door hem greedy te construeren. We gaan de modderpoelen van links naar rechts overdekken. Merk op dat het altijd goed is om een plank van lengte l zover mogelijk naar rechts neer te leggen, zolang die het meest linkse punt waar nog modder is overdekt. Het volgende stukje C-code geeft dus de oplossing voor het probleem.

```
int nplanken = 0, pos = 0;

for (int i=0; i<N; i++) {
    if (pos >= interval[i].einde) continue;
    pos = max(pos, interval[i].begin);
    int n = (interval[i].einde - from + L - 1) / L;
    nplanken += n;
    pos += n*L;
}
```

D Drugsbende

Probleem

Bondig geformuleerd is het probleem: gegeven een gesorteerde rij getallen (de cellen), vind een deelrij (waar de gevangenen geplaatst gaan worden), zodat het minimale verschil tussen twee opeenvolgende getallen van die deelrij zo groot mogelijk is.

Oplossing

Eerst merken we op dat het makkelijk is om te kijken of een bepaald kleinste verschil (noem dit D) bereikt kan worden. Het kleinste element uit de rij met cellen kun je natuurlijk altijd gebruiken. Vervolgens gebruik je steeds de kleinste die in ieder geval D van de vorige verwijderd ligt, totdat je K posities gehad hebt of geen posities meer over hebt.

Een C++-implementatie hiervan ziet er als volgt uit (hier is N het aantal cellen, de array $x[..]$ de posities van de cellen en is K het aantal te plaatsen gevangenen):

```
bool possible (int D) {
    int prev = x[0], i = 0;
    for (int k = 1; k<K; k++) {
        while (i<N && x[i]<prev+D ) i++;
        if ( i==N ) return false;
        prev = x[i];
    }
    return true;
}
```

Het is duidelijk dat deze code $O(N)$ tijd vergt.

Om nu het gewenste antwoord te vinden, voeren we een binary search uit op D. Dit levert in totaal een $O(N \log x_{\max})$-algoritme.

E Safari

Probleem

Al mag de herfst in Nederland alweer hebben ingezet, deze opgave neemt je mee naar het zonovergoten Zuid-Afrika. De vraag is namelijk wat de maximale kans is om een leeuw te zien tijdens een safari. Deze safari moet beginnen en eindigen in hetzelfde gegeven punt en de lengte ervan moet tussen bepaalde grenzen liggen. Als invoer krijgt je programma een lijst met kruispunten en de wegen daartussen, waarvan de lengte en kans om een leeuw te zien gegeven is.

Oplossing

We vragen ons eerst af wat de kans is om een leeuw te zien als je eerst een weg doorrijdt waarvoor de kans p_1 is en vervolgens een weg waarvoor de kans p_2 is. Elementaire kansrekening geeft dat dit $1-(1-p_1)(1-p_2)$ is[1]. Met behulp van deze wetenschap reduceert het probleem tot dynamic programming.

Hiertoe definiëren we $\texttt{mogelijk}[i,t]$ en $\texttt{best}[i,t]$ waarin we bijhouden of het mogelijk is om op tijdstip t op plaats i te zijn, en als dat het geval wat dan de maximale kans is om een leeuw te zien. Voordat we verder gaan, voeren we eerst nog wat notatie in. Voor een weg w noteren we het beginpunt met $w.b$, het eindpunt met $w.e$ en de kans om een leeuw te zien met $w.p$.

Als er een weg w is met $w.e = i$ zodat $\texttt{mogelijk}[w.b, t - w.l]$, dan is $\texttt{mogelijk}[i,t]$ waar.

Verder is

$$\texttt{best}[i,t] = \max_w (1 - \texttt{best}[w.b, t - w.l])(1 - w.p)$$

waarbij je het maximum neemt over alle wegen met $w.e = i$ en waarvoor $\texttt{mogelijk}[w.b, t - w.l]$ waar is.

Als je dit hebt uitgevoerd, mag het duidelijk zijn dat de gevraagde kans gelijk is aan $\max_t \texttt{best}[0,t]$ waarbij t tussen de aangegeven grenzen voor de lengte van de safari moet liggen en bovendien $\texttt{mogelijk}[0,t]$ waar moet zijn.

[1] Dit is als volgt in te zien: de kans om geen leeuw te zien is $1-p$. Als je op een route van twee wegen geen leeuw ziet, dan zie je dus op geen van beide wegen een leeuw, oftewel die kans is $(1-p_1)(1-p_2)$ en de kans om wel een leeuw te zien dus $1-(1-p_1)(1-p_2)$

F Parkeren

Probleem

Op een rechthoekige parkeerplaats willen we aan iedere bezem een parkeervak toekennen op zo'n manier dat de bezem die het verste moet vliegen, een zo klein mogelijk afstand moet overbruggen.

Oplossing

Het probleem zegt het bijna al, we willen een (bipartiete) matching vinden tussen de bezems en parkeervakken. We gaan ervanuit dat het bekend is hoe bipartiet matchen werkt. Sla hier desnoods een algoritmen-boek op na.

Construeer nu eerst een graaf met als vertices de bezems en parkeervakken en edges van de bezems naar de vakken met lengte gelijk aan de afstand tussen de twee.

De matching die we zoeken, moet aan het volgende voldoen: de langste edge die gebruikt wordt, moet zo klein mogelijk zijn. We kunnen deze matching op de volgende twee manieren vinden:

- Neem een bovengrens voor de lengte van de edges en kijk of je een matching kunt maken met alle bezems, waarbij je alleen edges gebruikt van lengte kleiner of gelijk aan de bovengrens. Linear zoeken op de bovengrens levert een $O(N^3(X+Y))$-algoritme en binair zoeken een $O(N^3 \log(X+Y))$-algoritme. (Hier is N het aantal bezems plus parkeervakken en zijn X en Y de dimensies van de parkeerplaats.)
- Sorteer de edges op volgorde en begin met een graaf zonder edges. Voeg steeds een edge aan de graaf toe en kijk of je de matching kunt uitbreiden. Zodra de matching uitgebreid is tot één waarbij alle bezems gebruikt worden, heb je het antwoord gevonden. Een simpele implementatie hiervan levert een $O(N^4)$-algoritme en met een handige datastructuur kan dit versneld worden tot $O(N^3)$.

Al deze vier algoritmen zouden de testdata door kunnen komen, al hoewel de $O(N^3(X+Y))$-variant krap wordt.

G Coalitie

Probleem

Het probleem bestaat per run uit twee delen: bepaal het aantal meerderheidscoalities en bepaal het aantal meerderheidscoalities dat kritisch is voor het eerste land.

Oplossing

Beide delen kunnen door middel van dynamisch programmeren opgelost worden.

De variabele die we bijhouden voor het eerste probleem is $c[s, b, i]$: het aantal meerderheidscoalities met s stemmen, bevolking b dat met de eerste i landen gevormd kan worden. Zij s_i het aantal stemmen van land i, b_i het aantal inwoners in land i, S het totale aantal stemmen, B het totale aantal inwoners en N het aantal landen. Voor $i = 0$ gelden de randvoorwaarden $c[0, 0, 0] = 1$ en $c[s, b, 0] = 0$ als $(s, b) \neq (0, 0)$.

Vervolgens bepalen we met de volgende recursie de waarden voor $i > 0$:

$$c[s, b, i + 1] = c[s, b, i] + c[s - s_i, b - b_i, i]$$

De data kunnen worden bijgehouden in een array van grootte $S \cdot B$. Het uitrekenen van de array $c(s, b, N)$ is van orde $S \cdot B \cdot N$. Nadat $c[s, b, N]$ bepaald is kan het aantal meerderheidscoalities bepaald worden door te sommeren over $c[s, b, N]$ met $s \geq 0.5S$ en $b \geq 0.6B$.

Voor het tweede deel houden we een soortgelijke variabele bij, namelijk $d[s, b, i]$: het aantal meerderheidscoalities met s stemmen, bevolking b dat met de eerste i landen gevormd kan worden waarbij het eerste land in de coalitie zit. Het oplossen van het tweede deel gaat analoog aan het eerste deel.

H Rondje om de wereld

Probleem

In de voetsporen van Jules Verne, vragen wij ons af of het mogelijk is om vanuit een gegeven stad één of meerdere rondjes om de wereld te reizen en weer bij ons beginpunt uit te komen. Hiertoe is een lijst van steden met hun lengte- en breedtegraad, en de vluchten die hiertussen mogelijk zijn gegeven. Vluchten lopen altijd langs de kortste weg over het aardoppervlak.

Oplossing

Een manier om dit probleem op te lossen, is door middel van een depth-first search. Naast dat je bijhoudt welke steden je bezocht hebt, houd je echter ook de hoek bij die je in totaal hebt afgelegd om daar te komen. Als je nu tijdens de depth-first search een stad voor de tweede keer aandoet en de gereisde hoek is hetzelfde, is dat oninteressant en hoef je ook niet verder te zoeken. Maar als de gereisde hoek verschillend is, kun je de twee verschillende paden naar die stad aan elkaar plakken om zo een rondje om de wereld te vinden.

Om de gereisde hoek te bepalen, moet je weten wanneer je rechtsom danwel linksom ten opzichte van de aardas reist. Het mag duidelijk zijn dat de breedtegraad hierbij totaal irrelevant is. Als je van een stad met lengtegraad ϕ_1 naar een stad met lengtegraad ϕ_2 reist, zien we dat:

- als $\phi_1 < \phi_2$ vlieg je rechtsom als $\phi_2 - \phi_1 < 180$
- als $\phi_1 > \phi_2$ vlieg je rechtsom als $\phi_2 - \phi_1 + 360 < 180$
- in alle andere gevallen vlieg je linksom.

Zie onderstaande C++ code.

```
void search (int n, int nowangle)
{
  if (angle[n]!=-999 && nowangle!=angle[n]) possible=true;
  if (angle[n]!=-999) return;

  angle[n]=nowangle;
  for (int i=0; i<loc[n].conn.size(); i++) {
    int next = loc[n].conn[i];
    int diff = loc[next].latt - loc[n].latt;
    if (diff > 180) diff = diff-360;
    if (diff < -180) diff = diff+360;

    search (next, nowangle+diff);
  }
}

void solve ()
{
  possible=false;
  angle=vector<int>(loc.size(), -999);
  search(0,0);
  cout << (possible ? "ja" : "nee") << endl;
}
```

I Back to BASIC

Probleem

Het probleem is duidelijk: we moeten programma schrijven dat regel voor regel een BASIC programma volgens de gegeven syntax uitvoert.

Oplossing

Dit lijkt op het eerste gezicht misschien een lastig parsing probleem, maar dat valt erg mee: alle statements zijn erg simpel en hebben een vaste vorm. De enige uitzondering hierop zijn expressies, die uit één waarde of twee waarden en één enkele operator kunnen bestaan en verder nog het `IF THEN` statement, wat weer door een ander (eventueel `IF THEN`) statement gevolgd kan worden. Dit laatste is geen probleem, omdat er geen `ELSE` is: na het testen, weten we of we het statement na `THEN` uit moeten voeren en hoeven we niet meer te letten op de voorafgaande `IF THEN`.

De oplossing kan het simpelst gemaakt worden, door bij te houden waar in het programma je op dit moment aan het uitvoeren bent en dan steeds het eerste "token" op de regel te lezen: is dit een commando, dan voer je dat uit, anders is het een variabele toekenning en moet je de expressie na de '=' parsen. Verder moet je de positie in het programma natuurlijk updaten afhankelijk van het commando.

Verder kun je een expressie inlezen, door het eerste token te lezen en daarna te kijken of er nog een token op de regel staat. Zo ja, dan is dat een operator en volgt nog een waarde. Van beide waarden moet natuurlijk gekeken worden op het een letter (dus variabele) is, danwel een getal.

De randvoorwaarden van het probleem zorgen ervoor dat je nooit tegen vreemde situaties aanloopt, maar het is natuurlijk wel aan te raden, om variabele waarden in iets groters dan een 16-bits variabele bij te houden.

De code om dit alles te parsen kan lineair geschreven worden, dus de maximale looptijd wordt de maximale regellengte keer het maximale aantal uitgevoerde statements en dat is zonder problemen binnen de tijd te doen.