Zesde college algoritmiek

14 maart 2013

Backtracking

Verdeel en Heers

Basisidee backtracking

- bouw een oplossing stap voor stap op en controleer steeds of de deeloplossing in conflict komt met de restricties (en nog wel tot een oplossing kan leiden)
- op elk moment kun je kiezen uit een aantal mogelijke vervolgstappen; maak een keuze en ga langs die weg verder met het opbouwen van de oplossing
- als een keuze op niets uitloopt, herzie je deze keuze en probeer je een andere mogelijkheid

Vergelijk

- het vinden van de uitgang in een doolhof: loop steeds verder en als je bij het zoeken vastloopt, ga terug op je pad om het laatste open alternatief te proberen

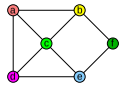
Genereer alle permutaties van 1 t/m n met backtracking.

```
void permutaties(int n, int perm[ ], int hierzo) {
    int i;
    if (hierzo == n+1)
        drukaf(perm,n); // permutatie gevonden
    else {
        for (i=1; i<=n; i++) {
            perm[hierzo] = i;
            if (!aanwezig(perm, i, hierzo))  // test of i
            // al in perm[1] t/m perm[hierzo-1] voorkomt
               permutaties(n, perm, hierzo+1);
        } // for
    } // else
} // permutaties
```

Vraag: wat heeft dit met torens op een schaakbord te maken?

Definitie: Een Hamiltonkring in een (ongerichte) graaf is een kring die elke knoop precies één keer aandoet.

Voorbeeld: a b f e c d a is een Hamiltonkring in nevenstaande graaf, echter a b c d e f a is geen Hamiltonkring.



Probleem: vind een Hamiltonkring in een gegeven ongerichte graaf.

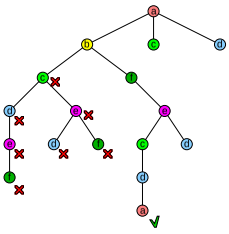
Exhaustive search: genereer alle $(n-1)!$ kandidaatoplossingen (permutaties van de knopen) en controleer daarna van elk of het een Hamiltonkring voorstelt.

Backtracking: genereer de mogelijke Hamiltonkringen stap voor stap* en controleer tijdens de constructie al of de deeloplossing wel aan de restricties† voldoet.
Kies in elke uitbreidingsstap steeds de eerstvolgende **buurknoop** (in een of andere volgorde) en controleer hiervan of deze nog niet geweest is in het reeds geconstrueerde deelpad. Blijkt die keuze toch (hier of verderop in de constructie) op niets uit te lopen, kies dan de volgende buurknoop. Als er geen buurknopen meer zijn kan het deelpad blijkbaar niet meer uitgebreid worden en moet je je vorige keuze herzien.

*hier: tak voor tak of knoop voor knoop

†alle knopen verschillend; tak tussen opeenvolgende knopen

Een beschrijving van de werking van het backtracking-algoritme voor het vinden van een Hamiltonkring in de voorbeeldgraaf wordt gegeven door de volgende state space tree:



. breid het pad telkens met één knoop uit (hier: keuzes in alfabetische volgorde)

. uitbreidingen met knopen die al eerder voorkwamen in het pad zijn niet weergegeven

. in de knopen met een rood kruis backtrackt het algoritme zodra blijkt dat die niet tot een oplossing leiden

Probleem:
Gegeven een verzameling $S = \{s_1, s_2, \ldots s_n\}$ van positieve (> 0) gehele getallen en een geheel getal d. Laat S oplopend gesorteerd zijn. Vind een deelverzameling (of alle deelverzamelingen) van S waarvan de som der getallen gelijk is aan d.

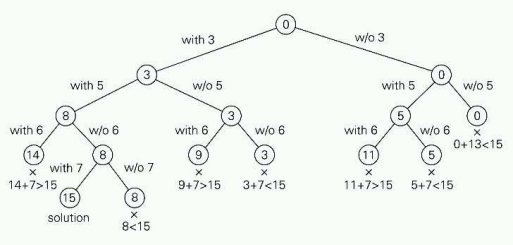
Voorbeelden:
$S = \{1, 2, 5, 6, 8\}$ en $d = 9$. Er zijn twee oplossingen, namelijk $\{1, 2, 6\}$ en $\{1, 8\}$.
$S = \{3, 5, 6, 7\}$ en $d = 15$. Er is één oplossing: $\{3, 5, 7\}$.

Exhaustive search: genereer alle 2^n deelverzamelingen van S en controleer of hun som gelijk is aan d.

De stap-voor-stap constructie van deeloplossingen doen we (bijvoorbeeld) zo: gegeven een deeloplossing (= een veelbelovende deelverzameling van $\{s_1, s_2, \ldots, s_i\}$), dan zijn er twee mogelijke vervolgstappen: óf s_{i+1} wordt toegevoegd, óf s_{i+1} wordt niet toegevoegd.

Backtracking bekijkt zo ook alle deelverzamelingen, maar hoeft ze niet allemaal volledig te genereren. Een (veelbelovende) deelverzameling van $\{s_1, s_2, \ldots, s_i\}$ met som s' hoeft niet verder uitgebreid te worden als $s' + s_{i+1} > d$ (*), of als $s' + \Sigma_{j=i+1}^{n} s_j < d$.

Opmerking:
Als (*) geldt levert elke uitbreiding, met welke s_j ($j \geq i+1$) dan ook een te grote totaalsom op. Dus herzie je vorige keuze. Hier is gebruikt dat S oplopend gesorteerd is.



Volledige state-space tree bij toepassing van backtracking op probleeminstantie $S = \{3, 5, 6, 7\}$ en $d = 15$ (waarbij alle oplossingen gezocht worden). De knopen stellen deeloplossingen voor. Het getal in een knoop is de som van de s_j uit de corresponderende deelverzameling. De ongelijkheid onder een blad geeft aan waarom daar backtracking plaatsvindt.

Knapzakprobleem

Gegeven n objecten, met gewicht $w_1, \ldots, w_n$ en waarde $v_1, \ldots, v_n$, en een knapzak met capaciteit W. **Gevraagd**: de meest waardevolle deelverzameling der objecten die in de knapzak past (dus met totaalgewicht $\leq W$).

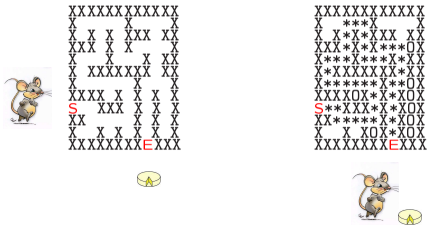
Voorbeeld:

object	gewicht	waarde
1	8	42
2	3	14
3	4	40
4	5	27

knapzakcapaciteit 12

- genereer de deelverzamelingen **stap voor stap**, bijvoorbeeld door steeds aan een goede deelverzameling van de objecten 1 t/m i achtereenvolgens object $i+1$ of $i+2$ of $\cdots$ n toe te voegen (mogelijke **keuzes**)
- **controleer** of het totaalgewicht van de aldus uitgebreide verzameling nog steeds $\leq W$ is
- zo nee, **herzie** dan **je keuze** (en probeer het volgende object)
- zo ja, ga dan op **dezelfde** manier verder
- houd ook de totaalwaarde van de (deel)verzamelingen bij en de tot dusver gevonden maximale waarde

Gegeven een rechthoekig doolhof. Gevraagd wordt een pad van Start naar Eind, waarbij alleen horizontaal en verticaal gelopen mag worden.




```
bool dwaal(int x, int y) {
// is er een pad van (x,y) naar (x_eind,y_eind) ?
  int richting, x_volgende, y_volgende;
  if ( ( x == x_eind ) && ( y == y_eind ) ) { // gevonden!
    doolhof[x][y] = '*';
    return true;
  }
  else if ( doolhof[x][y] != ' ' ) { // geen vrije plek
    return false; }
  else {
    doolhof[x][y] = '?'; // tijdelijk markeren; voorkomt ∞ loopen
    for ( richting = OOST; richting <= NOORD; richting++ ) {
      x_volgende = volgende_x(x,richting);
      y_volgende = volgende_y(y,richting);
      if ( dwaal(x_volgende,y_volgende) ) {
        doolhof[x][y] = '*';
        return true;
      }
    }
    doolhof[x][y] = 'O'; // afgehandeld:
    return false;        // geen rechtstreeks pad via deze (x,y)
  }
}
```


Laten oplossingen van een bepaald probleem van de vorm $(X[1], X[2], \ldots, X[m])$ zijn en zij S_i de verzameling waarden die $X[i]$ kan aannemen. De algemene vorm van een backtracking algoritme is dan:

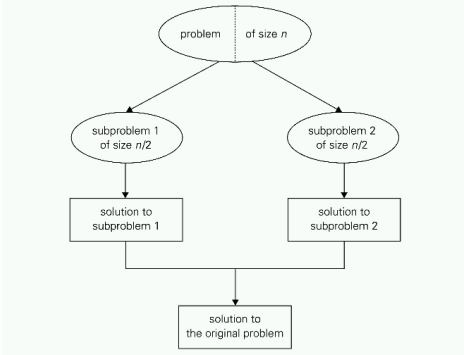
backtrack$(X[1 \cdots i])$::
// $X[1 \cdots i]$ is een veelbelovende deeloplossing, consistent met
// de restricties; we zoeken *alle* oplossingen
if $X[1 \cdots i]$ is een oplossing **then**
  print$(X[1 \cdots i])$;
else
  for elke $x \in S_{i+1}$ consistent met $X[1 \cdots i]$ en de restricties **do**
    $X[i+1] := x$;
    backtrack$(X[1 \cdots i+1])$;
  od
fi

Divide and Conquer

1. Verdeel een instantie van het probleem in twee (of meer) kleinere instanties
2. Los de kleinere instanties op: meestal recursief
3. Combineer deze twee (of meer) oplossingen tot een oplossing van de oorspronkelijke (grotere) instantie

Opmerking: meestal wordt een probleeminstantie in twee ongeveer gelijke delen verdeeld.

Verdeel en heers (vaak: verdeel in twee gelijke delen)

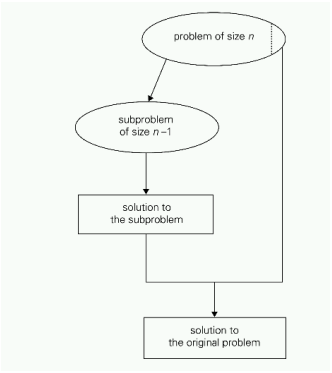


Decrease and Conquer

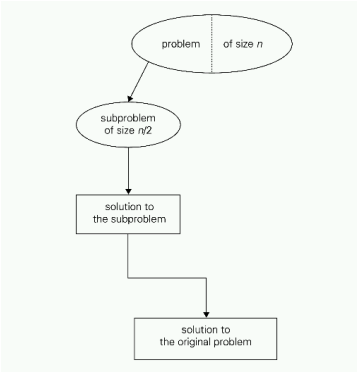
1. Reduceer een instantie van het probleem tot een kleinere instantie van hetzelfde probleem
2. Los de kleinere instantie op: vaak recursief
3. Breid de oplossing van de kleinere probleeminstantie uit tot een oplossing van de oorspronkelijke instantie

In het boek wordt onderscheid gemaakt tussen:
- Decrease by one
- Decrease by a constant factor
- Variable-size decrease

Decrease by one



Decrease by a constant factor (decrease by half)



Verdeel en heers en sorteren:

Sorteer(rij)::

if (de rij heeft meer dan één element) **then**

Verdeel de rij in twee stukken: linkerrij en rechterrij;

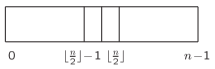
Sorteer(linkerrij);

Sorteer(rechterrij);

Combineer linkerrij en rechterrij;

fi .

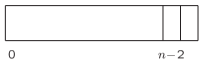
Divide and conquer



Mergesort

Quicksort

Decrease and conquer (decrease by one)



Insertion sort

Mergesort($A[0 \cdots n-1]$)::

// sorteert het array $A[0..n-1]$ recursief

// uitvoer: $A[0..n-1]$ oplopend gesorteerd

if $n > 1$

copieer($A[0 \cdots \lfloor \frac{n}{2} \rfloor - 1], B[0 \cdots \lfloor \frac{n}{2} \rfloor - 1]$);

copieer($A[\lfloor \frac{n}{2} \rfloor \cdots n-1], C[0 \cdots \lceil \frac{n}{2} \rceil - 1]$);

Mergesort($B[0 \cdots \lfloor \frac{n}{2} \rfloor - 1]$);

Mergesort($C[0 \cdots \lceil \frac{n}{2} \rceil - 1]$);

Merge(B, C, A);

fi .

Merge($B[0 \cdots p-1], C[0 \cdots q-1], A[0 \cdots p+q-1]$) ::
// voegt 2 gesorteerde arrays B en C samen tot 1 gesorteerd array A
$i, j, k := 0$;
// voeg samen totdat een van de twee op is: ritsen
while $i < p$ **and** $j < q$ **do**
if $B[i] \leq C[j]$ **then**
$A[k] := B[i]; k := k+1; i := i+1;$
else
$A[k] := C[j]; k := k+1; j := j+1;$
od
// en de rest
if $i = p$ **then**
copieer $C[j \cdots q-1]$ naar $A[k \cdots p+q-1]$;
else
copieer $B[i \cdots p-1]$ naar $A[k \cdots p+q-1]$;
fi .

Quicksort($A[l \cdots r]$)::

// sorteert het (sub)array $A[l \cdots r]$ recursief

// uitvoer: $A[l \cdots r]$ oplopend gesorteerd

if $l < r$

s :=Partitie($A[l \cdots r]$); // s het splitspunt

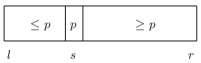
Quicksort($A[l \cdots s-1]$);

Quicksort($A[s+1 \cdots r]$);

fi .

Partitie($A[l \cdots r]$) ::
// partitioneert een (sub)array, met $A[l]$ als spil (pivot)
$p := A[l]$;
$i := l$; $j := r + 1$;
repeat
repeat $i := i + 1$; **until** $i > r$ **or** $A[i] \geq p$;
repeat $j := j - 1$; **until** $A[j] \leq p$;
if $i < j$ **then**
Wissel($A[i], A[j]$);
if
until $i \geq j$;
Wissel($A[l], A[j]$);
return j; .

Partitie



Probleem: reorganiseer de elementen van een gegeven array A zodanig dat alle negatieve elementen voorafgaan aan de positieve. Het algoritme moet lineair zijn en in situ. Hint: vergelijk Partitie.

```
i = 0; j = n-1;
while (i <= j) {
    if (A[i] < 0)
        i = i+1;
    else {
        wissel(A[i],A[j]);
        j = j-1;
    }
}
```

Variant (Dutch National Flag): gegeven een array met 'R', 'W' en 'B'. Reorganiseer het array zodat v.l.n.r. eerst alle 'R', dan de 'W' en dan de 'B' staan. Zie Levitin, opgave 5.2.9.a.

Insertionsort($A[0 \cdots m-1]$)::
if $m > 1$
Insertionsort($A[0 \cdots m-2]$);
Voeg $A[m-1]$ op de juiste plek in;
fi .

Invoegen van $A[m-1]$ in het reeds gesorteerde voorstuk $A[0] \cdots A[m-2]$ door van rechts naar links $A[m-1]$ te vergelijken met $A[i]$. Deze recursieve versie komt overeen met de iteratieve versie zoals bij Programmeermethoden behandeld (zie ook Levitin):

$A[0] \leq A[1] \leq \ldots \leq A[i] \leq A[i+1] \leq \cdots \leq A[m-3] \leq A[m-2] || A[m-1] \cdots$
kleiner of gelijk $A[m-1]$ ↑ hier invoegen groter dan $A[m-1]$

Mergesort:
- worst case complexiteit: $\Theta(n \log n)$
- extra geheugen: $O(n)$

Quicksort:
- worst case complexiteit: $\Theta(n^2)$ voor (o.a.) het reeds gesorteerde rijtje
- average case complexiteit: $\Theta(n \log n)$
- extra geheugen: in situ

Insertion sort:
- worst case/average case complexiteit: $\Theta(n^2)$
- extra geheugen: in situ

- **Lezen/leren bij dit college:**
  Paragraaf 12.1, 5 inl., 5.1, 5.2, 4 inl., 4.1
- **Werkcollege:**
  donderdag 14 maart 2013, 13:45–15:30, in zaal Noordeinde
  donderdag 4 april 2013, 13:45–15:30, Paleistuin:
  tweede programmeeropdracht
- **Opgaven:**
  http://www.liacs.nl/home/rvvliet/algoritmiek/
- **Volgend college:**
  donderdag 4 april 2013