

---

## Programmeermethoden

Algoritmen

week 13: 5–9 december 2011

<http://www.liacs.nl/home/kosters/pm/>

Op allerlei colleges en in allerlei boeken en artikelen worden **algoritmen** behandeld, bijvoorbeeld bij de volgende Informatica-colleges in Leiden (semester tussen haakjes):

- Programmeermethoden (1), Algoritmiek (2)
- Datastructuren (3), Kunstmatige intelligentie (4)
- . . .

Je kunt algoritmen op allerlei manieren rubriceren, bijvoorbeeld met behulp van de volgende begrippen:

verdeel en heers, numerieke wiskunde, graafalgoritmen, dynamisch programmeren, patroonherkenning, adversary, data mining, geometrisch modelleren, backtracking, benaderende algoritmen, kunstmatige intelligentie, neurale netwerken, evolutionaire algoritmen,  $P \leftrightarrow NP$ , gretige algoritmen, snelle Fouriertransformatie,

...

We bekijken er een paar van.

Als  $n$  een priemgetal is, geldt dat  $a^{n-1} - 1$  deelbaar is door  $n$  voor  $a = 1, 2, \dots, n - 1$ . Het omgekeerde is bijna waar.

Dit suggereert het volgende algoritme dat “bepaalt” of  $n$  een priemgetal is: Als  $2^{n-1} - 1$  niet deelbaar is door  $n$  is  $n$  zeker geen priemgetal, en anders (misschien) wel. Dit gaat fout bij 341, 561,  $\dots$ , maar dat is te verbeteren: probeer andere  $a$ ; echter, 561, een Carmichael-getal, blijft lastig. (Uiteindelijk: Miller-Rabin.)

Het algoritme is een **randomized** algoritme, en wel een **Monte Carlo** algoritme: het ene antwoord is altijd juist, het anders soms niet. Bij **Las Vegas** algoritmen zijn de antwoorden altijd juist — maar het duurt soms lang.

En hoe maak je een willekeurige **permutatie**, dat wil zeggen, een random volgorde van de getallen  $1, 2, \dots, n$ ?

```
// stop random permutatie van 0,1,...,n-1 in array A
void maakpermutatie (int A[ ], int n) {
    int i; // array-index
    int r; // random array-index
    for ( i = 0; i < n; i++ ) A[i] = i;
    for ( i = n-1; i >= 0; i-- ) {
        r = rand ( ) % ( i+1 ); // 0 <= r <= i
        wissel (A[i],A[r]);
    }//for
}//maakpermutatie
```

rand ( ) geeft een random-getal; srand (42) zet het “seed”.

Een **gretig** (greedy) algoritme neemt beslissingen door één stap vooruit te kijken; het zijn meestal **benaderende** algoritmen.

We bekijken het volgende algoritme voor het **Common Superstring probleem**, dat vraagt naar een (zo kort mogelijke) string die een stel gegeven strings bevat: Neem herhaald de twee meest overlappende strings bij elkaar.

Een voorbeeld. Begin met TCAGT, CATCAG, GTG en GCA. De twee meest overlappende strings zijn CATCAG en TCAGT; vervang deze door CATCAGT, we hebben dan CATCAGT, GTG en GCA over.

Zowel GTG als GCA hebben een overlap van 2 met CATCAGT. Kies bijvoorbeeld GTG, wat CATCAGTG en GCA oplevert.

De eindoplossing is GCATCAGTG, en die is toevallig optimaal.

Nog een voorbeeld: begin met GCC, ATGC en TGCAT.

De twee meest overlappende strings zijn ATGC en TGCAT; vervang deze door ATGCAT, en we houden ATGCAT en GCC over.

Deze twee strings hebben geen overlap, dus de eindoplossing is hun “concatenatie”: ATGCATGCC of GCCATGCAT, beide van lengte 9. De optimale oplossing, TGCATGCC, heeft echter lengte 8!

Het algoritme vindt wel snel een superstring, maar niet altijd een optimale . . .

Algemener: begin met

$$\{C(AT)^k, (TA)^k, (AT)^kG\}$$

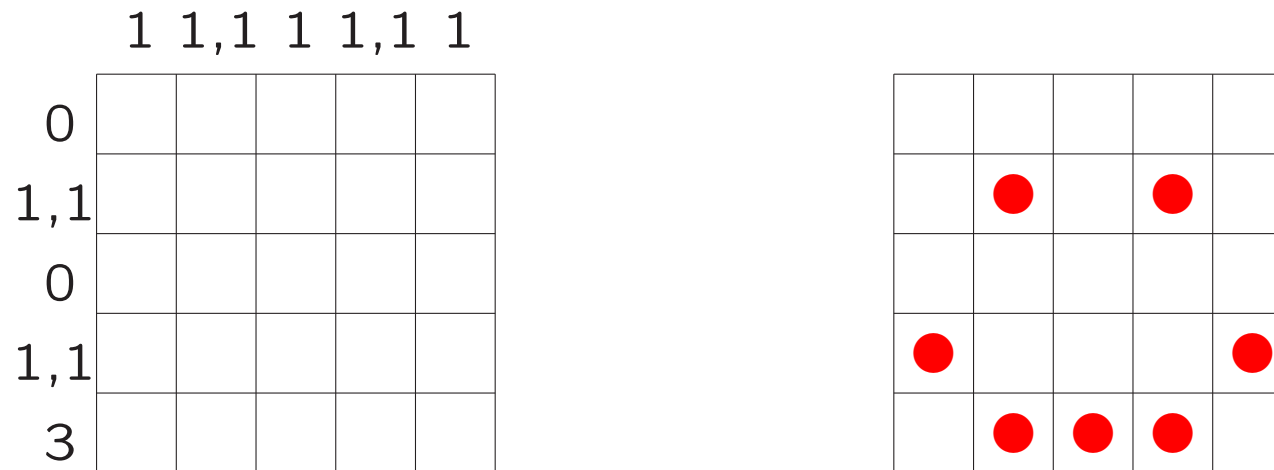
voor een vaste  $k \geq 1$ , dan levert het algoritme  $C(AT)^kG(TA)^k$  ter lengte  $4k+2$ , terwijl de optimale string  $C(AT)^{k+1}G$  lengte  $2k+4$  heeft.

De uitvoer van het gretige algoritme kan dus twee keer zo lang zijn dan de optimale. Maar erger wordt het niet (open problemen).

Dit soort algoritmen wordt gebruikt bij DNA-reconstructie: de “shotgun-methode”.

**Genetische algoritmen**, of algemener **Evolutionaire algoritmen**, evolueren een populatie met kandidaat-oplossingen voor een probleem. Van elke kandidaat-oplossing kun je de kwaliteit berekenen met een **fitness-functie**. De beste individuen gaan door, en met **mutatie** (willekeurige, kleine veranderingen) en **crossover** (combineer twee “ouders”) krijg je een nieuwe generatie.

Japanse puzzels (Nonogrammen) zien er zo uit:



Naast iedere rij en boven iedere kolom staan in volgorde de lengtes van aaneengesloten series **rode** blokjes.

<http://www.liacs.nl/home/kosters/nono/>

Genetische algoritmen kunnen gebruikt worden om Nonogrammen op te lossen.

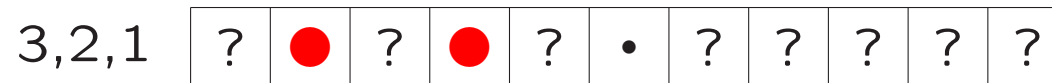
Een individu is hier een string (of array) met 25 bits (algemeener, voor een  $m$  bij  $n$  puzzel, met  $mn$  bits), waarbij een 1 rood en een 0 leeg voorstelt. Je kunt er voor kiezen om het aantal enen per rij altijd “goed” te houden.

Mutatie zou bijvoorbeeld in een rij een 1 en een 0 kunnen verwisselen. Als je handig muteert kun je “alles” bereiken!

De fitness-functie is een som over rijen en kolommen. Per rij/kolom geef je aan hoeveel je van de specificatie afwijkt — en dat is nog lastig precies te maken.

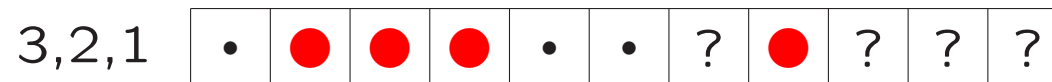
**Dynamisch Programmeren** kan gebruikt worden om Nonogrammen op te lossen.

De vraag is wat bij een lijn (rij of kolom) kan worden afgeleid, gegeven een deelinvulling:



Een • betekent een zeker leeg vakje, een ● staat voor een zeker gevuld vakje. De rest is nog onbekend.

Dan concluderen we:



Hoe helpt **Dynamisch Programmeren** hierbij?

In plaats van alle manieren te bedenken waarmee de lijn kan worden gevuld (en te kijken wat deze gemeenschappelijk hebben) kun je ook tabellen maken om te zien hoe dit zit met deelrijen en deelbeschrijvingen!

We berekenen  $L_{ij}$ : kan de (deel)beschrijving  $d_1, d_2, \dots, d_i$  worden gerealiseerd in de (deel)string  $s_1, s_2, \dots, s_j$ ; en zo ja: wat “moet”? Hierbij geldt  $0 \leq i \leq \text{lengte beschrijving}$  en  $0 \leq j \leq \text{lengte string}$ .

**Maxi** en **Mini** spelen het volgende eenvoudige spel: **Maxi** wijst eerst een horizontale rij aan, en daarna kiest **Mini** een verticale kolom.

3	12	8
2	4	6
14	5	2

Bijvoorbeeld: **Maxi** kiest rij 3, daarna kiest **Mini** kolom 2; dat levert einduitslag 5.

**Maxi** wil graag een zo groot mogelijk getal, **Mini** juist een zo klein mogelijk getal.

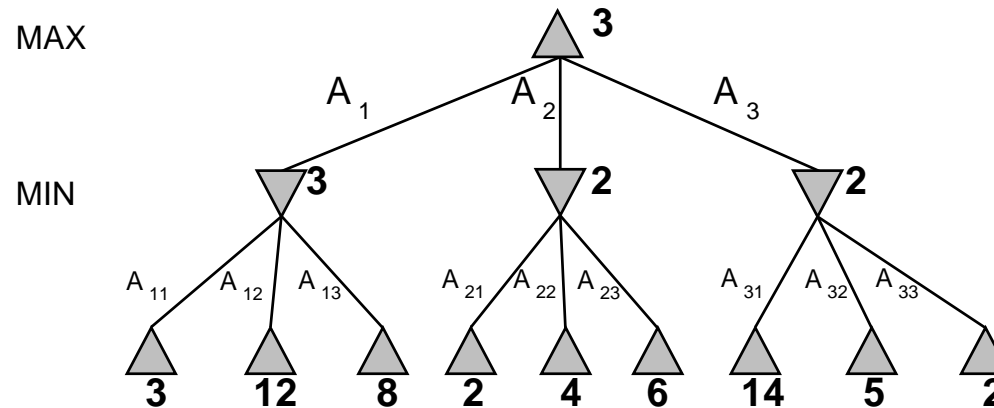
Hoe analyseren we dit spel?

Als **Maxi** rij 1 kiest, kiest **Mini** kolom 1 (levert 3); als **Maxi** rij 2 kiest, kiest **Mini** kolom 1 (levert 2); als **Maxi** rij 3 kiest, kiest **Mini** kolom 3 (levert 2). Dus kiest **Maxi** rij 1! Dit heet een **brute force** redenering: we hebben echt alles bekeken.

3	12	8
2	?	?
14	5	2

Nu merken we op dat de analyse (het **minimax-algoritme**) hetzelfde verloopt als we niet eens weten wat onder de twee vraagtekens zit. Het  **$\alpha$ - $\beta$ -algoritme** onthoudt als het ware de beste en slechtste mogelijkheden, en kijkt niet verder als dat toch nergens meer toe kan leiden.

In boomvorm:



Het **minimax-algoritme** is “recursief”: neem in bladeren de evaluatie-functie, in MAX-knopen het maximum van de kinderen, in MIN-knopen het minimum van de kinderen. MAX- en MIN-knopen wisselen elkaar af.

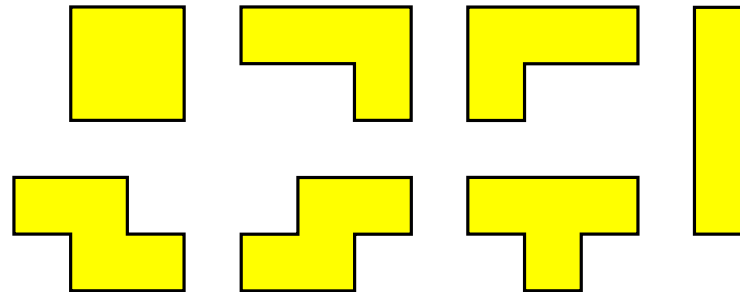
Bovenstaande boom is **één zet** (= move) diep, oftewel **twee ply**.

Aan een spel als **Tetris** kleven allerlei vragen:

- Hoe speel je het zo goed mogelijk?  
(AI = Kunstmatige intelligentie)
- Hoe moeilijk is het? (complexiteit)
- Wat kan er allemaal gebeuren?

Zo is bijvoorbeeld bewezen dat sommige Tetris-problemen **NP-volledig** zijn (gezamenlijk werk met mensen van MIT), dat je bijna alle configuraties kunt bereiken, maar dat niet alle problemen “beslisbaar” zijn.

De 7 Tetris-stukken:



De vraag “Kun je met een gegeven serie (inclusief volgorde) van deze stukken een deels al gevuld bord helemaal leeg spelen?” is NP-volledig.

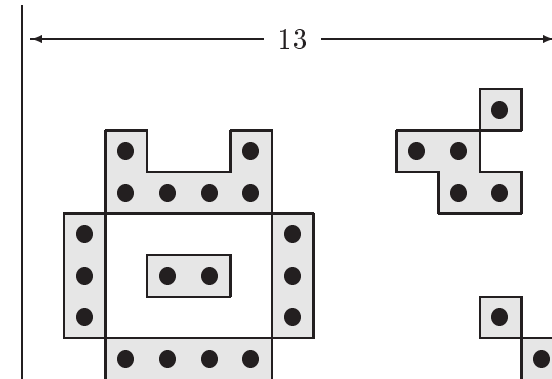
Als iemand het bord leeg speelt kun je dat eenvoudig controleren. Als het *niet* kan, kan men (tot nu toe) niks beters verzinnen dan alle mogelijkheden één voor één na te gaan!

P en NP zijn “klassen” van **beslissingsproblemen**. Het probleem of een graaf samenhangend is, zit in P. Het probleem of een graaf een Hamilton-circuit heeft, zit in NP, en is zelfs **NP-volledig**; je kunt het “eenvoudig”, maar niet efficiënt, oplossen met bruteforce technieken.

P is de klasse van beslissingsproblemen die door een “deterministische Turing-machine” in “polynomiale tijd” kunnen worden opgelost. NP is de klasse van beslissingsproblemen die door een “niet-deterministische Turing-machine” in “polynomiale tijd” kunnen worden opgelost: je mag “gokken”.

Open probleem: geldt  $P = NP$ ?

Een “willekeurige” configuratie:



Deze kan gemaakt worden door 276 geschikte Tetris-stukken op de juiste plaats te laten vallen.

Claim: op een bord van oneven breedte kan elke configuratie bereikt worden!

<http://www.liacs.nl/home/kosters/tetris/>

- werk aan de vierde programmeeropgave — de deadline is op 9 december 2011 → werkcolleges/vragenuren
- college Java/Qt/...: dinsdag 13 december
- dinsdag 13 en donderdag 15 december 2011: werkcollege met oude tentamens, zalen B1 en B2
- **vragenuur over tentamen**: dinsdag 3 januari 2012, 11.00 uur, Snellius zaal 174
- **tentamen**: woensdag 4 januari 2012, 10.00–13.00 uur, Snellius (hertentamens: 28 maart, 30 juli 2012)