# Uitwerking tentamen Algoritmiek

9 juli 2014 10:00 – 13:00

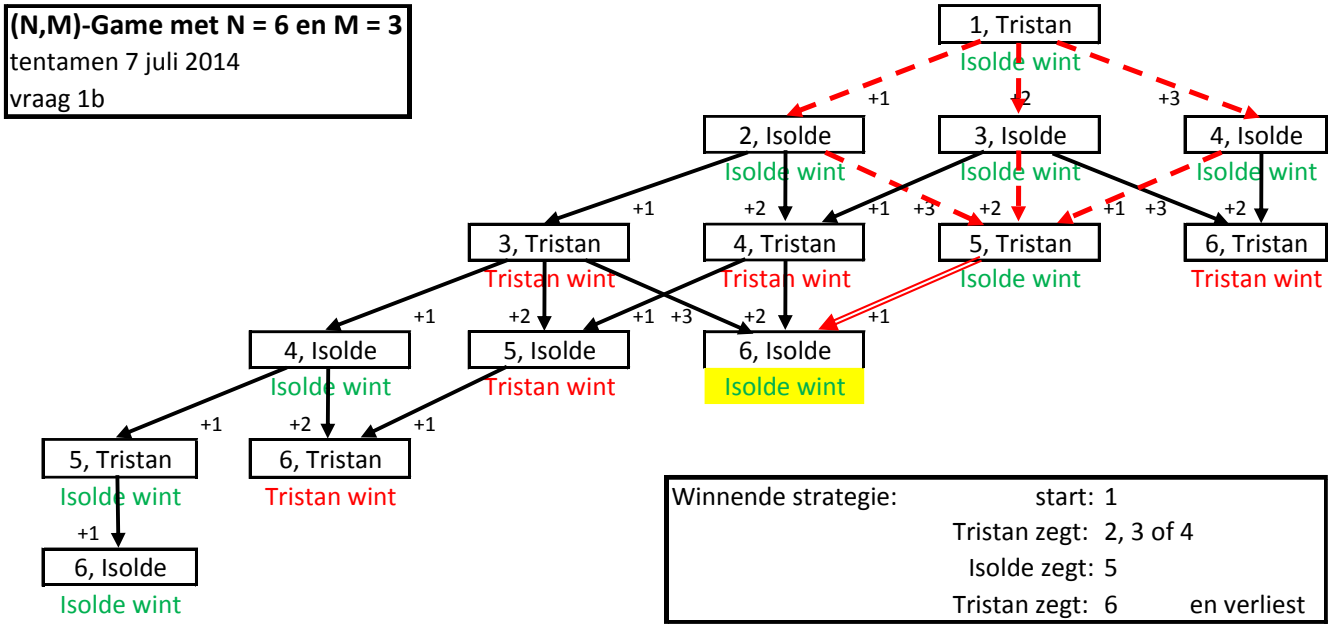
## 1. (N,M)-game

a. Toestanden: Een geheel getal g, waarvoor geldt $1 \leq g \leq N$ én wie er aan de beurt is (Tristan of Isolde)

Acties: Het noemen van een geheel getal h, waarvoor geldt $g < h \leq g + M$ door de speler die aan de beurt is

Eindtoestanden: (N, Tristan) en (N, Isolde)

b.

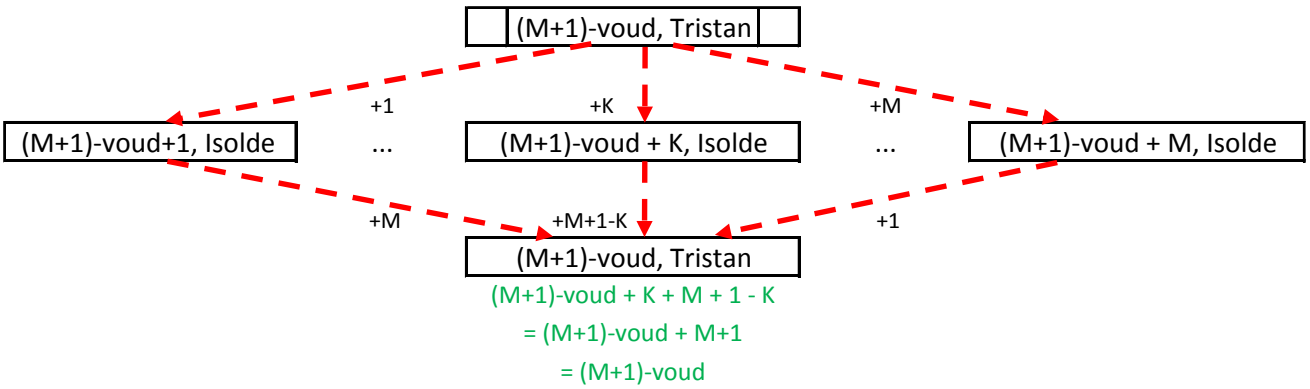



c.

(i) Het zeggen van een (M + 1)-voud is een winnende strategie:
Als Tristan kans ziet om N-1 te zeggen, speelt hij Isolde in dwang, en kan zij alleen nog N zeggen. Dit zelfde geldt natuurlijk ook voor Isolde. Omdat N een (M+1)-voud+1 is, is N-1 een (M+1)-voud.
Wanneer een speler een (M+1)-voud zegt kan de ander dit onmogelijk ook doen. Want een (M+1)-voud + K is alleen een (M+1)-voud als K zelf een (M+1)-voud is. Echter geldt dat $1 \leq K \leq M$. K kan dus nooit een (M+1)-voud zijn.

Het verschil tussen twee opeenvolgende (M+1)-vouden is M+1. We hebben net aangetoond dat dit verschil niet overbrugd kan worden door één actie. Dus als Tristan telkens een (M+1)-voud kan zeggen, kan Isolde dat niet. Tristan kan dan op een gegeven moment het (M+1)-voud N-1 zeggen. Waarna Isolde N moet zeggen en verliest.

(ii) Tristan kan deze winnende strategie uitvoeren en Isolde niet:
Omdat het startgetal 1 is, kan Tristan altijd 1+M zeggen, wat meteen het eerste (M+1)-voud is.
Het verschil met het volgende (M+1)-voud is telkens M+1. Dit verschil kan nooit met één actie overbrugd worden (zie(i)). Het verschil M+1 kan echter wel altijd door twee acties overbrugd worden: Als de ene speler met K verhoogt, Kan de andere speler met L = M+1-K verhogen. Omdat $K \in [1..M]$ is ook $L = M+1-K \in [1..M]$. ($K=1 \Rightarrow L=M$ en $K=M \Rightarrow L=1$) . Tristan kan dus inderdaad altijd een (M+1)-voud zeggen.



## 2. Binaire Boom

a.
```
void som( knoop* wortel )
{
  if ( wortel != NULL )                    // er is een knoop (i.v.m. lege boom)
  {
    wortel->som = wortel->info;            // initialisatie van som op info waarde van knoop
    if ( wortel->links != NULL )           // er is een linker kind
    {
      som( wortel->links );                // eerst recursief som berekenen in subboom
      wortel->som += wortel->links->som;   // dan deze som optellen bij som huidige knoop
    }
    if ( wortel->rechts != NULL )          // er is een rechter kind
    {
      som( wortel->rechts );               // eerst recursief som berekenen in subboom
      wortel->som += wortel->rechts->som;  // dan deze som optellen bij som huidige knoop
    }
  }
}
```

b.
```
bool balans( knoop* wortel )
{
  if ( wortel == NULL )
    return true;                           // een lege knoop is in balans
  else
    if ( wortel->links != NULL             // check of de knoop 2 kinderen heeft
         && wortel->rechts != NULL         // en of huin som-veld ONgelijk is
         && wortel->links->som             // we gebruiken short circuit evaluation
              != wortel->rechts->som )
      return false;                        // de knoop en de boom zijn NIET in balans
    else
      return                               // de knoop is in balans
        ( balans( wortel->links )          // de boom is in balans als beide subbomen
        && balans( wortel->rechts ) );     // in balans zijn.
}
```

3. **Dijkstra** tentamen 9 juli 2014 vraag 3

Afstanden tussen de knopen

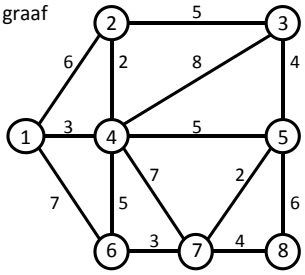
| to | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 1 | - | 6 | - | 3 | - | 7 | - | - |
| 2 | 6 | - | 5 | 2 | - | - | - | - |
| 3 | - | 5 | - | 8 | 4 | - | - | - |
| 4 | 3 | 2 | 8 | - | 5 | 5 | 7 | - |
| 5 | - | - | 4 | 5 | - | - | 2 | 6 |
| 6 | 7 | - | - | 5 | - | - | 3 | - |
| 7 | - | - | - | 7 | 2 | 3 | - | 4 |
| 8 | - | - | - | - | 6 | - | 4 | - |

Uitvoerig Dijkstra's Algoritme

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | min | U | Actie |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | | 1 | Start bij knoop 1 |
| - | 6 | ∞ | 3 | ∞ | 7 | ∞ | ∞ | 3 | 4 | Kies knoop 4 vanaf knoop 1 |
| - | 5 | 11 | - | 8 | 7 | 10 | ∞ | 5 | 2 | Kies knoop 2 vanaf knoop 4 |
| - | - | 10 | - | 8 | 7 | 10 | ∞ | 7 | 6 | Kies knoop 6 vanaf knoop 1 |
| - | - | 10 | - | 8 | - | 10 | ∞ | 8 | 5 | Kies knoop 5 vanaf knoop 4 |
| - | - | 10 | - | - | - | 10 | 14 | 10 | 3 | Kies knoop 3 vanaf knoop 2 * |
| - | - | - | - | - | - | 10 | 14 | 10 | 7 | Kies knoop 7 vanaf knoop 4 * |
| - | - | - | - | - | - | - | 14 | 14 | 8 | Kies knoop 8 vanaf knoop 5 |

* deze stappen mogen worden omgewisseld

Originele graaf



Legenda:

nr — De kolom U bevat de knopen, die in elke stap worden toegevoegd aan U

nr — De lengte van het minimale pad wat aan de boom wordt toegevoegd

Knopen en takken van de originele graaf die nog niet bekeken zijn

nr — De definitieve lengte van het minimale pad naar de betreffende knoop

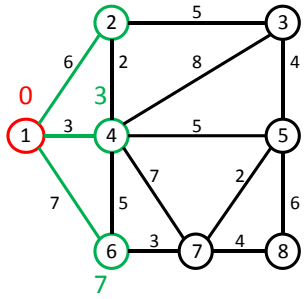
Knopen uit U en de takken die tot de kortste paden boom van U horen

nr — De lengte van het minimale pad naar de betreffende knoop

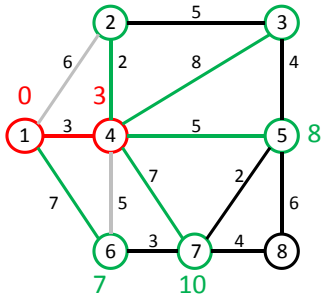
Kandidaat knopen voor de selectie van de volgende knoop

Tak die definitief niet tot de kortste paden behoort

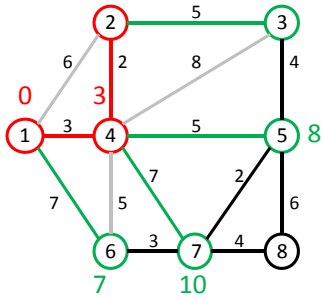
Stap 1



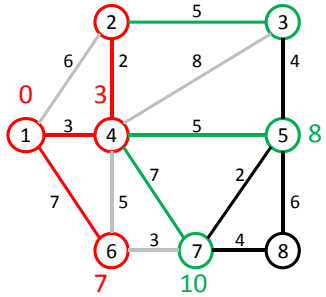
Stap 2



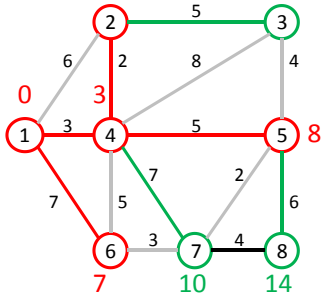
Stap 3



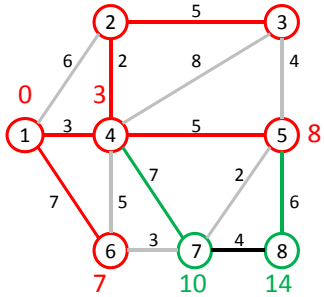
Stap 4



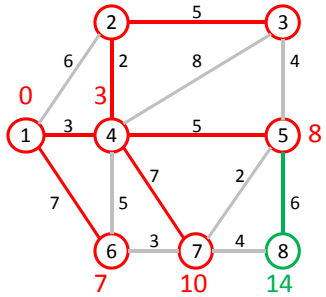
Stap 5



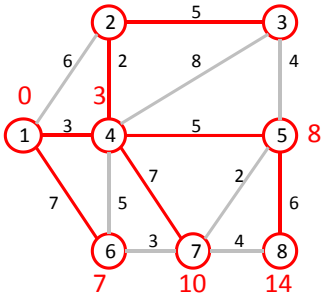
Stap 6



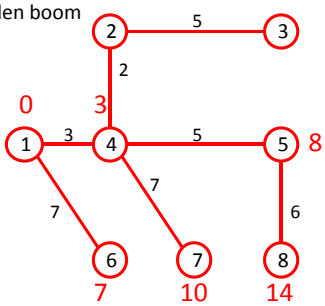
Stap 7



Stap 8



kortste paden boom



## 4. Dubbel Nul

a.
```
bool bevat00( char A[], int n )
{                                       \\ loop alle elementen van A langs
  for ( int i=0; i<n-1; i++ )           \\ n-1 i.v.m. vergelijking A[i] én A[i+1]
  {
    if ( A[i] == '0'                    \\ vergelijk twee naast elkaar gelegen elementen
         && A[i+1] == '0' )             \\ wanneer beide 0 bevatten zijn we klaar
      return true;                      \\ A bevat 00
  }
  return false;                         \\ wanneer we alle elementen bezocht hebben
}                                       \\ en nog geen 00 vonden, bevat het array geen 00
```

b.
```
bool dubbelnul( char A[], int i )
{
  if ( i <= 0)                          \\ een array met 1 element of minder
    return false;                       \\ kan geen 00 bevatten
  else
    if ( A[i] == '0'                    \\ als de laatste twee elementen van A beide 0 zijn we klaar
         && A[i-1] == '0' )             \\ het array bevat 00
      return true;
    else                                \\ Als we geen 00 vonden,
      return dubbelnul( A, i-1 );       \\ bekijken we de twee op één na laatste elementen
}                                       \\ d.m.v. een recursieve aanroep, zonder het laatste element.
```

c. In beide gevallen is een rij met enkel énen het worst case scenario (ook afwisselend 0 en 1). Dit is afhankelijk van de exacte implementatie bij a en b.
Algoritme a geeft 2(n-1) vergelijkingen (n-1 keer de for-loop. Per iteratie 2 vergelijkingen ( `A[i] == '0' && A[i+1] == '0'` )
Algoritme b geeft 2(n-1) vergelijkingen (n recursieve aanroepen tot de stop-conditie wordt bereikt. Per aanroep 2 vergelijkingen ( `A[i] == '0' && A[i-1] == '0'` ). Alle beid de laatste recursieve aanroep (als i = 0), worden deze vergelijkingen niet gedaan.
Het aantal vergelijkingen bij beide algoritmen is dus gelijk.
Het iteratieve algoritme heeft nu de voorkeur omdat hier niet voor elke recurie stap de state van de functie op de stack hoeft worden op geslagen.

d.
```
bool nulnul( char A[], int links, int rechts )
{
  if ( links >= rechts)                 \\ een array met 1 element of minder
    return false;                       \\ kan geen 00 bevatten
  else
  {
    int mid = (links + rechts - 1) / 2; \\ bepaal de middelste 2 elementen van A
    if ( A[mid] == '0'                  \\ wanneer beide 0 bevatten zijn we klaar
         && A[mid+1] == '0' )           \\ A bevat 00
      return true;
    else
      return                            \\ Als we geen 00 vonden, bekijken we van A
        ( nulnul( A, links, mid )       \\ de linker en rechter helft via recursie.
        || nulnul( A, mid+1, rechts ) );
  }                                     \\ door short cirquit evaluation,
}                                       \\ wordt de rechter helft alleen bekeken
                                        \\ wanneer de linker helft géén 00 bevat
```

## 5. Spartaanse Bruiloft

a. Bij best-fit-first branch and bound bouwen we oplossingen stap voor stap op.

Bij iedere deeloplossing die we genereren, berekenen we onmiddelijk een bovengrens (het is een maximalisatie probleem) voor de waarde die een complete oplossing kan hebben die uit deze deeloplossing voortkomt (**bound**).

Tijdens het algoritme pakken we steeds de deeloplossing met de hoogste bovengrens (**best-fit-first**) en werken die één stap verder uit. Dat wil zeggen: we breiden de deeloplossing op alle mogelijke manieren één stap uit naar grotere deeloplossingen (**branch**).

Bij alle nieuwe deeloplossingen berekenen we natuurlijk weer een bovengrens.

We breiden een deeloplossing niet verder uit:

- Als we zien dat er geen geldige complete oplossing uit voort kan komen (de deeloplossing is **infeasable**, **snoeien**).
- Als er nog precies één complete oplossing uit voort kan komen. In dat geval construeren we deze complete oplossing, berekenen haar echte waarde (geen bovengrens) en als deze waarde hoger is dan de hoogste tot nu toe gevonden echte waarde, onthouden we die.
- Als de bovengrens van de deeloplossing tot nu toe ≤ hoogste tot nu toe gevonden echte waarde. Dan kan de deeloplosing geen hogere waarde meer opleveren (**snoeien**).

b. We kiezen per stap een vrouw (op volgorde van de rijen in de tabe) en breiden de huidige deeloplossing uit met de verschillende paren die deze vrouw nog kan vormen met de mannen, die nog niet gekoppeld zijn in de deeloplossing onde beschouwing. (**branch**) (We genereren dan geen **infeasable** oplossingen)

Als **bovengrens** kiezen we de succeswaarde van de tot nu toe gekoppelde paren plus voor elke nog niet gekoppelde vrouw (elke rij dus) de hoogste score waarbij alleen gekeken wordt naar mannen, die nog niet gekoppeld zijn. Elek vrouw wordt in ieder geval gekoppeld aan één van deze mogelijke mannen. Elke vrouw kan maximaal de ze hoogste score bijdragen aan de score. Elke vrouw kan geen hogere score bijdragen aan de totaalscore. De succeswaarde kan niet hoger worden, dan het totaal van deze maxima opgeteld bij de al berekende succeswaarde van de deeloplossing.

c.