Uitwerking tentamen Algoritmiek

10 juni 2014 10:00 – 13:00

1. Dominono's

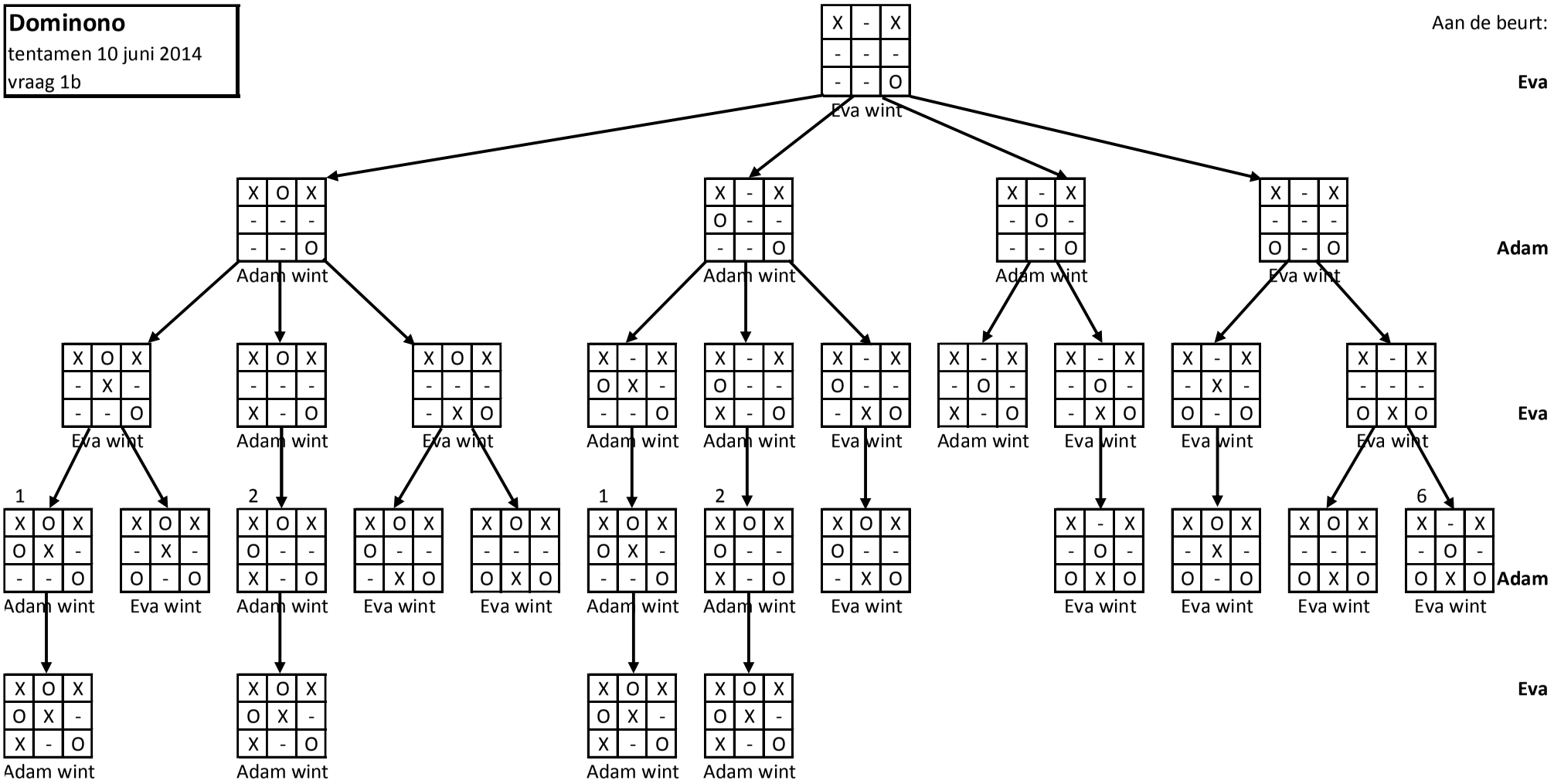
a. Toestanden: *n* x *n* bord met in elk hokje een O, een X of een -. Hierbij is het aantal X gelijk aan het aantal O of hooguit één hoger. Bovendien bevatten twee horizontaal of verticaal aangrenzende hokjes nooit beide een X of beide een O. De speler die aan de beurt is volgt uit het aantal X-en en O's.

Acties: Vervang in een van de hokjes een - door een X voor Adam, door een O voor Eva. Hierbij mag geen X of O geplaatst worden, wanneer een horizontaal of verticaal aangrenzend hokje dat zelfde symbool al bevat.

b. Twee mogelijkheden voor linksboven:

- stel daar staat een X.
  Dan moet daarnaast en eronder een O staan, en daarnaast/onder weer een X, enzovoort. Zo krijg je per twee rijen afwisselend X O X O X O X en volgende rij O X O X O X O.
  - n is oneven: je hebt per twee rijen evenveel X en O. Echter, aangezien er een oneven aantal rijen is eindig je met X O X O X O X, dus heb je dan altijd 1 X meer dan een O. Dat is mogelijk.
  - n is even: Je hebt per rij evenveel O als X en in totaal ook evenveel O als X.
- stel nu dat er een O linksboven staat.
  Dan vanwege hetzelfde argument als hierboven: afwisselend O en X.
  - n is oneven: een oplossing op met in totaal 1 O meer dan er X-en staan op. Dat kan niet, omdat X begon.
  - n is even: er is wel een vol bord dat met O linksboven begint.

Dominono
tentamen 10 juni 2014
vraag 1b



Eva wint.
Winnende strategie:

X	-	X
-	-	-
O	-	O

hierna leidt elke legale zet tot winst voor Eva

De cijfers geven aan welke toestanden gelijk zijn. Deze hoeven maar één keer verder uitgewerkt te worden. Voor de andere toestand kan de winnaar worden overgenomen van de toestand die al is uitgewerkt.

2. Binaire Boom

a.
```
void ouders( knoop* wortel )
{
  if ( wortel != NULL )                    // er is een knoop
  {
    if ( wortel->links !=NULL )            // er is een linker kind
    {
      wortel->links->vader = wortel;       // knoop is de vader van het linker kind
      ouders( wortel->links );             // recursieve aanroep van ouders voor linker kind
    }
    if ( wortel->rechts !=NULL )           // er is een rechter kind
    {
      wortel->rechts->vader = wortel;      // knoop is de vader van het rechter kind
      ouders( wortel->rechts );            // recursieve aanroep van ouders voor linker kind
    }
  }
}
```

b.
```
void herfst( knoop* wortel )
{
  if ( wortel != NULL )                    // met een lege boom mogen we niks doen
    if ( wortel->links == NULL             // check of knoop een blad is
      && wortel->rechts == NULL            // we gebruiken short circuit evaluation
      && wortel->vader != NULL )           // de knoop moet een vader hebben
    {
      if ( wortel->vader->links == wortel )// het is een blad:
        wortel->vader->links = NULL;       //   kijk of de knoop het linker of het
      else                                 //   rechter kind is van zijn vader en
        wortel->vader->rechts = NULL;      //   verwijder deze uit de boom
      delete wortel;                       //   geef ook het gebruikte geheugen vrij
    }
    else
    {                                      // het is geen blad:
      herfst( wortel->links );             //   recursief aanroepen van herfst voor
      herfst( wortel->rechts );            //   het linker en het rechter kind
    }
}
```

c.
```
int neven( knoop* p )
{                                          // in de vraag staat dat de knoop p bestaat
  int aantalNeven = 0;                     // een test op p != NULL is dus overbodig
  knoop* opa, oom;
  if ( p->vader != NULL                    // heeft de knoop een vader en een opa
       && p->vader->vader != NULL )        // we gebruikenshort circuit evaluation
  {
    opa = p->vader->vader;
    if ( opa->links == p->vader )          // als de vader het linker kind is van opa
      oom = opa->rechts;                   // dan is oom het rechter kind
    else
      oom = opa->links;                    // anders is oom het linkerkind van opa
    if ( oom != NULL )                     // als er een oom is zijn er neven
    {
      if ( oom->links != NULL )            // oom heeft een linker kind: dat is een neef
       aantalNeven++;
      if (oom->rechts != NULL )            // oom heeft een rechter kind: dat is een neef
       aantalNeven++;
    }
    return aantalNeven;
}
```

3. Bergwandeling

a. Gretige strategie: Start in (0, 0). Kijk in elke hectare naar het hoogteverschil met de hectare in oostelijke richting en dat met de hectare in zuidelijke richting. Kies de richting met het kleinste hoogteverschil.

0	3	5	6	5	4
1	2	2	2	7	3
1	8	7	6	6	2
3	9	7	4	5	3
7	8	7	5	6	2
4	5	8	4	5	0

Gretige route:

- zuid, zuid, zuid, zuid, zuid, oost, oost, zuid, oost, oost, oost (ZZZZOOZOOO).
- (0, 0), (1, 0), (2, 0), (3, 0), (4, 0), (4, 1), (4, 2), (5, 2), (5, 3), (5, 4), (5, 5).

Het hoogteverschil van deze route is: 1 + 0 + 2 + 4 + 1 + 1 + 1 + 4 + 1 + 5 = 20.

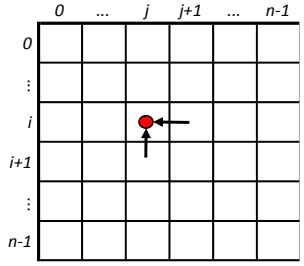
b. M[n - 1][n - 1] = 0: Als je in (n-1,n-1) bent hoef je niet meer verder te lopen (kan ook niet); het (minimale) hoogteverschil met (n-1,n-1) is dan uiteraard 0.

M[n-1][j] = M[n-1][j+1] + abs(H[n-1][j+1] – H[n-1][j]): Als je in (n-1,j) bent kun je nog maar 1 kant op, namelijk alleen oostwaards (naar rechts). Het minimale hoogteverschil dat je kunt halen is dan het minimale hoogteverschil vanaf je oostelijke buur (n-1,j+1) (dus M[n-1][j+1]) plus het (absolute) hoogteverschil dat je overbrugt door van (n-1,j) naar (n-1,j+1) te lopen.

M[i][n-1] = M[i+1][n-1] + abs(H[i+1][n-1] – H[i][n-1]): Vergelijkbaar met boovenstaande, maar voor zuidwaarts.

M[i][j] = min(M[i+1][j] + abs(H[i+1][j] – H[i][j]), M[i][j+1] + abs(H[i][j+1] – H[i][j])): Dit is het algemene geval. Vanuit (i,j) kun je alleen naar de hectare direct ten oosten (rechts) of naar de hectare direct ten zuiden (onder), dus naar (i,j+1) of naar (i+1,j). Van daaruit is het minimale hoogteverschil van een wandeling naar (n-1,n-1) gelijk aan M[i][j+1] resp. M[i+1][j]. Dus, als je vanuit (i,j) via de rechterbuur naar (n-1,n-1) wil, is het minimale hoogteverschil dat je kan bereiken gelijk aan M[i][j+1] + het hoogteverschil abs(H[i][j] - H[i][j+1]) tussen (i,j) en zijn rechter buur (i,j+1). Iets soortgelijks voor een wandeling die via de benedenbuur (i+1,j) naar (n-1,n-1) gaat. Het minimale hoogteverschil om van (i,j) naar (n-1,n-1) te komen is dan natuurlijk het minimum van deze twee mogelijkheden.

c. M[i][j] wordt bepaald uit M[i+1][j] en M[i][j+1]. Deze moeten dus bepaald worden voor M[i][j]. Vullen van M moet daarom van rechtsonder naar links boven gevuld worden. Zowel de kolommen als de rijen woorden van n-2 terug naar 0 doorlopen. Dus:



- Eerst M[n-1][n-1]
- Dan de laatste rij (i = n-1) en de laatste kolom (j = n-1) voor i respectievelijk j van n-1 dalend naar 0. Laatste rij eerst of laatste kolom eerst maakt niet uit.
- Tenslotte in een dubbele loop met i én j van n-1 dalend naar 0. Per kolom of per rij, maakt niet uit.

```
M[n-1][n-1] = 0;
for ( int i = n – 2; i >= 0; i-- )
  M[i][n-1] = M[i+1][n-1] + abs( H[i+1][n-1] – H[i][n-1] );
for ( int j = n – 2; j >= 0; j-- )
  M[n-1][j] = M[n-1][j+1] + abs( H[n-1][j+1] – H[n-1][j] );
for ( int i = n – 2; i >= 0; i-- )
  for ( int j = n – 2; j >= 0; j-- )
    M[i][j] = min( M[i+1][j] + abs( H[i+1][j] – H[i][j] ),
                   M[i][j+1] + abs( H[i][j+1] – H[i][j] ) );
return M[0][0];
```

12	11	9	8	7	6
11	10	10	10	7	5
15	8	7	6	6	4
15	11	9	6	5	3
11	10	9	7	6	2
14	13	10	6	5	0

d. Het minimale hoogteverschil van de route is M[0][0] = 12. De bijbehorende route is:

- zuid, oost, oost, oost, zuid, oost, zuid, oost, zuid, zuid (ZOOOZOZOZZ).
- (0, 0), (1, 0), (1, 1), (1, 2), (1, 3), (2, 3), (2, 4), (3, 4), (3, 5), (4, 5), (5, 5).

Start in (0, 0) en herhaal tot de oostelijke of zuidelijke rand bereikt is (i = n-1 of j = n-1):

- De waarde in M[i][j] is berekend volgens de recurrente betrekking en is (van wege de min-functie) gelijk aan óf M[i+1][j] + abs(H[i+1][j] – H[i][j]), de route via (i+1, j) naar het zuiden, óf M[i][j+1] + abs(H[i][j+1] – H[i][j]) de route via (i, j+1) naar het oosten, of beide.
  - als M[i][j] – abs(H[i+1][j] – H[i][j]) = M[i+1][j]: ga naar het zuiden (i++)
  - anders: ga naar het oosten (j++) (dan is nl. M[i][j] – abs(H[i][j+1] – H[i][j]) = M[i][j+1])

Wanneer i = n – 1 of j = n – 1 is er nog maar één richting mogelijk en hoeft er niet meer gepuzzeld te worden.
N.B. Er is een versnelling mogelijk door op te merken, dat de route in M nooit van een hokje met een lagere waarde naar een hokje met een hogere waarde kan gaan.

4. Trakteren

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**).

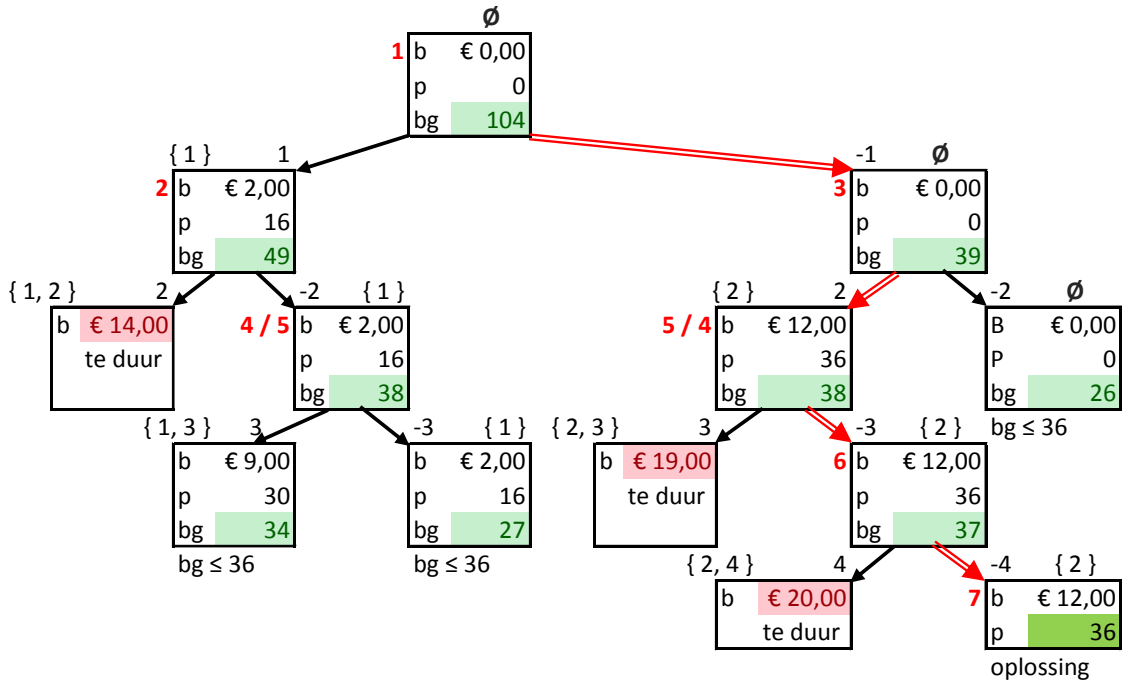
Bij alle nieuwe deeloplossingen berekenen we natuurlijk weer een bovengrens.

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. *p/e* is het maximale aantal porties per euro dat je kan krijgen door één taart te kopen. De taarten staan aflopend gesorteerd op *p/e*. p_1/e_1 is daardoor het gunstigst in aanschaf. Met een budget van *B* euro kan je dus maximaal $B * (p_1/e_1)$ porties kopen en nooit meer dan dat, want elke andere taart levert een slechter verhouding op.

Na stap i, is al bepaald voor elke taart 1 t/m i of deze taart t_i in de deeloplossing (verzameling gekochte taarten) voorkomt of niet. Deze deeloplossing heeft *b* euro gekost. Er is dus nog *(B – b)* euro te besteden. Het maximaal nog mogelijke aantal porties per euro is nu (p_{i+1}/e_{i+1}) van taart t_{i+1}. Dat betekent dat we voor de euro's die we nog kunnen besteden maximaal $(B – b) * (p_{i+1}/e_{i+1})$ porties kunnen kopen. Wanneer we dit optellen bij *p*, het aantal porties in huidige de deeloplossing, dan krijgen we het maximale aantal porties dat een complete oplossing kan bevatten die van deze deeloplossing wordt afgeleid.



c. Optimale oplossing koop alleen de boterkoek (36 porties; € 12,-; taart 2)

Toelichting:

- De rode cijfers geven de volgorde aan waarin de state space tree wordt uitgebreid.
- Wanneer b groter is dan B dan is een deeloplossing te duur.
- Na stap 7 is er een oplossing gevonden die 36 porties bevat. Alle deeloplosingen die niet tot een groter aantal porties kunnen leiden worden daarna gesnoeid (bg≤36).
- De gevonden deeloplossing blijkt uiteindelijk optimaal te zijn. De oplossing bestaat uit 36 porties boterkoek. De verzameling bevat dus alleen taart 2.