Eerste opgave van Algoritmiek: Tegelspel

Jens van der Sloot
S4018494

Arthur van der Sterren
S4097769

21 maart 2024

1 Introductie spel + opdracht

Dit programma is gebouwt rondom een tegelspel. Dit tegelspel bestaat uit 3 onderdelen; een pot (1 rij met tegels), meerdere schalen (kortere rijen met tegels) en een persoonlijk bord per speler (2d veld met tegels). Er bestaan 2 kleuren tegels, de speler die de meeste rijen vol met een kleur krijgt wint.

Elke rij van je veld mag maar 1 kleur in, om dit te vullen pak je alle tegels van 1 kleur uit een schaal naar keuze, deze moeten allemaal in dezelfde rij terechtkomen. Maar het aantal nieuwe tegels uit de schaal moet wel samen met je huidige aantal tegels passen in je rij, anders is het geen valide zet. De schaal wordt hierop aangevuld vanuit de pot (pakt de eerste N van de pot, met N = missende tegels uit de schaal en plaatst het op de plekken van de verdwenen karakters).

In dit programma wordt eerst de mogelijkheid gegeven een spel in te lezen wat informatie bevat over een spel; namelijk de inhoud van de pot, het aantal schalen met bijhorende aantal tegels per schaal, het aantal rijen wat een speler heeft en hoeveel vakjes er in een rij kan en ten slotte wie er aan de beurt is. Indien het geen geldige informatie bevat of niet kan worden geopent krijg je hier terugkoppeling over.

Hierna zijn verschillende functies beschikbaar voor gebruikers. Een functie die vectoren genereert met mogelijke verschillende zetten, een functie om een zet te doen (of ontdoen). Maar ook een bruteforce functie die de optimale zet berekend (besteScore), en nog een functie (bepaalGoedeZet) die op basis van de Monte Carlo methode en een aantal simulaties de best gevonden zet returned. Dit is dus niet de absoluut beste zet aangezien er hier met een max aantal simulaties gewerkt wordt en dus ongelijk aan besteScore. Ten slotte zijn er nog 2 opties: je kan bepaalGoedeZet (speler1) tegenover besteScore (speler2) zetten en hun een potje laten spelen, en je kan een experiment doen wat eerst met behulp van bepaalGoedeZet tot een eindpositie gaat, om vervolgens achteruit terug te gaan en te meten hoelang besteScore over elke iteratie doet.

2 besteScore

besteScore is een functie die de beste zet voor de huidige speler bepaalt met behulp van recursie. De functie neemt in twee variabelen in: `pair<int, char> &besteZet` en `long long &aantalStanden`. Met behulp van besteZet wordt steeds opgeslagen wat de beste eerste zet is, en deze is call by reference zodat je als je eerder op een andere plek deze gedeclareerd heb na het aanroepen van de functie kan je er gelijk mee verder rekenen, of printen naar de console zoals gedaan wordt in main.cc. De tweede, aantalStanden houdt bij hoeveel iteraties er vergaan zijn en is call by referance zodat na elke iteratie van recursie de nieuwe iteratiesw bij het vorige aantal iteraties gaan.

Het eerste wat gebeurt wanneer de functie besteScore aangeroept wordt is dat er gekeken wordt of de huidige positie geen eindstand is, als dit niet het geval is zal de code doorgaan en moeten de spelers dus nog optimale

zetten maken. Als dit wel het geval is is het spel klaar (voor een bepaalde richting, ofwel de vertakking van de boom heeft een einde bereikt) en dan moet nu de eindscore berekend worden voor de huidige speler, wat gedaan wordt met de functie berekenScore (die makkelijk het aantal volle rijen telt voor elke speler en het van elkaar aftrekt op basis van huidigeBeurt) waaropeenvolgens de functie de juiste score returned. Wat de volgende stap is met deze teruggestuurde score wordt wordt verderop uitgelegd.

Hoe de functie de zogenaamde beste zet voor elke speler bepaald werkt dus op basis van recursie, op elke positie ofwel vertakking van de boom wordt gekeken welke zetten valide zijn met behulp van bepaalVerschillendeZetten(), hieropvolgens wordt met een for loop door elk van deze zetten gelopen. En bij elke iteratie gaat de boom een vertakking verder: Er wordt dus een nieuwe kopie gemaakt van het bord (met behulp van *this op TegelSpel) en een van de zetten uit bepaalVerschillendeZetten wordt gedaan met behulp van doeZet. Op deze manier wordt verder gerekend totdat een eindnode en dus de eindconditie: eindstand (= ofwel speler heeft gewonnen of geen zetten) is bereikt. Hierop volgt dus een return van de score van deze positie, die wordt vergeleken met besteScore, een integer die eerder in de functie is gedefinieerd als INT_MAX en zich voordoet als temp. Nu wordt met de eerder teruggegeven score gekeken of: |`score`| < |`besteScore`|, en zoja wordt dan de zojuist geretourneerde score opgeslagen in `besteScore`:

$$\texttt{besteScore} = \texttt{score}$$

Vergeet niet om ook de beste zet te wijzigen:

$$\texttt{besteZet} = \texttt{zet}$$

De reden waarom er met absolute waarden wordt gewerkt bij het vergelijken, is dat de geretourneerde score een minteken bevat.

$$\texttt{int score} = -\texttt{kopie.besteScore(volgendeZet, aantalStanden)}$$

Dit samen met het gebruik van absoluut in de vergelijking is zeer belangrijk, aangezien een eindpositie (ofwel een endnode) door zowel speler 1 als speler 2 kan worden bereikt, en beide spelers hun optimale optimale zet moeten spelen. Bijvoorbeeld, als de mogelijke eindscores -1, 0, 1 en 2 kunnen zijn, dan moet de eindscore 0 zijn als beide spelers optimaal spelen. Omdat -1 kleiner is dan 0, zal de score -1 als eindscore worden doorgegeven als je hier geen absolute waarde gebruikt. Dit zou niet correct zijn, want dan heeft speler 1 onjuist gespeeld. Het minteken zorgt ervoor dat na elke zet de huidige speler om wordt gezet van “maximizer“ in “minimizer“; de beste zet voor de ene speler is de slechtste verwachte zet voor de tegenstander. Na het doorlopen van alle eindtakken zal dus de juiste eindscore wat wordt gezien als optimaal spel teruggestuurd worden en zal de optimale zet juist staan in de meegegeven: `pair<int, char> &besteZet`.

Ten slotte het laatste onderdeel van de functie besteScore is `aantalStanden`. aantalStanden wordt bij elke iteratie in de for loop: `for (auto zet : mogelijkeZetten)` geupdate: `aantalStanden++;` en zal aan het einde van de functie juist staan in de meegegeven: `long long &aantalStanden`.

De specifieke “oplossingsmanier“ waarvoor wij hebben gekozen bij de besteScore functie is een vorm van het Minimax algoritme, wat dus een recursief algoritme is. We hadden eigenlijk drie redenen waarom het logisch leek om deze methodiek toe te passen:

- Ten eerste: het tegelspel bevat 2 spelers.
- Ten tweede: het was een vereiste voor de besteScore functie dat beide spelers optimaal moesten spelen.
- Ten derde: hoe de score berekend wordt in het tegelspel “schreeuwd“ eigenlijk om een toepassing van het Minimax; bij Minimax is een speler dus de “maximizer“ en de ander de “minimizer“ en om te switchen van een goede score voor de een naar een goede score voor de ander is er niks anders dan een ontkenning (ofwel minteken) nodig.

In termen van efficiëntie is dit algoritme niet snel en besparen we dus nergens tijd uit, aangezien het elke mogelijke optie doorkijkt op zoek naar de beste zet voor elke. Echter, dit zorgt er wel voor dat we eenvoudig de absoluut beste score en beste zet vinden voor de huidige speler vanaf een gegeven positie.

3 resultaten

3.1 spel1.txt

Tijdens het aanroepen van besteScore op spel1.txt gegeven dat spel1.txt het volgende bevat:

```
gbbggbbggbgbbbggbgbgbbgbgg\n
2 4\n5 5\n4 0\n2 0\n1 0\n0 0\n0 3\n0 3\n0 4\n1 0\n2 0\n5 0\n0
```

en dat besteScore op de startpositie aangeroepen wordt is de door ons bereikte outkomst:

```
Maak een keuze: 4

Beste score is: 0
Een beste zet is: (0,g)
We hebben hiervoor 10019 standen bekeken.
Dit kostte 5989 clock ticks, ofwel 0.005989 seconden.
```

besteScore geeft ofwel dus aan dat de beste zet (0,g) is en dat als beide spelers optimaal spelen de eindscore 0 zal zijn. Ook staat in de cout hoeveel standen bekeken zijn en over hoeveel clock ticks dit gebeurde. Daarnaast is het ook wel leuk om het aantal bekeken standen per clock tick te berekenen. In dit geval was dat:

$$\frac{10019}{5989} = 1.673 \text{ standen bekeken per clock tick} \tag{1}$$

Als we deze uitkomst vergelijken met bepaalGoedeZet zien we dat bepaalGoedeZet over 10 keer handmatig proberen 10 keer dezelfde uitkomst geeft. Hieruit zou je ofwel een conclusie kunnen stellen dat dit spel niet ingewikkeld genoeg is en dat in een “simpel“ geval bepaalGoedeZet even goed zal werken als besteScore.

Ten slotte keken we naar bepaalGoedeScore, de functie die een spel simuleert tussen twee spelers, waarbij speler1 de ”domme”speler is waarbij de zet bepaald wordt door bepaalGoedeZet en speler2 de ”slimme”speler die bepaald wordt door besteScore. Het 10x van de functie aanroepen geeft de volgende resultaten:

```
Score goed tegen best is: 0
```

Hieruit volgt dus eigenlijk dezelfde conclusie als uit de vorige alinea: spel1 is er niet ingewikkeld genoeg voor dat de algoritmen een verschil hebben in hoe goed ze het spel kunnen spelen.

3.2 spel3.txt

Tijdens het aanroepen van besteScore op spel3.txt gegeven dat spel3.txt het volgende bevat:

```
bbbbbgbbgbbbgbgbbbggbggbbggbggbbgbgbbbg\n
5 5\n10 6\n3 0\n6 0\n1 0\n0 4\n0 6\n0 6\n0 4\n1 0\n2 0\n6 0\n0 2\n
0 6\n0 1\n0 2\n3 0\n5 0\n6 0\n4 0\n4 0\n6 0\n1
```

en dat besteScore op de startpositie aangeroepen wordt is de door ons bereikte outkomst:

```
Maak een keuze: 4

Beste score is: 0
Een beste zet is: (0,b)
We hebben hiervoor 1356123 standen bekeken.
Dit kostte 1273184 clock ticks, ofwel 1.27318 seconden.
```

besteScore geeft ofwel dus aan dat de beste zet (0,b) is en dat als beide spelers optimaal spelen de eindscore 0 zal zijn. Ook staat in de cout hoeveel standen bekeken zijn en over hoeveel clock ticks dit gebeurde. Daarnaast als we dus het aantal standen per clock tick berekenen zien we:

$$\frac{1356039}{1273184} = 1.065 \text{ standen bekeken per clock tick} \tag{2}$$

Ofwel bijna 60 procent slomer dan bij spel1! Als we een gok zouden moeten wagen waarom het slomer zou zijn, leek het logisch dat het lag aan het feit dat er bij spel3 veel meer schalen zijn dan bij spel1 en dus hoogstwaarschijnlijk meer verschillende zetten die door de functie bepaalVerschillendeZetten verwerkt moeten worden, want deze functie wordt vaak aangeroepen in besteScore.

Als we deze uitkomst vergelijken met bepaalGoedeZet zien we dat bepaalGoedeZet over 10 keer handmatig proberen 6 keer voor de juiste zet (0,b) koos en 4 keer voor (1,b) koos. In eerste instantie klinkt het raar dat er verschillende antwoorden zijn uit dezelfde functie over meerdere keren aanroepen, maar dit is logisch, aangezien bepaalGoedeZet werkt op basis van Monte Carlo:

```
pair<int,char> randomZet = zetten[randomGetal(0, zetten.size() - 1)]
```

Ten slotte keken we naar bepaalGoedeScore, de functie die een spel simuleert tussen twee spelers, waarbij speler1 de ”domme”speler is waarbij de zet bepaald wordt door bepaalGoedeZet en speler2 de ”slimme”speler die bepaald wordt door besteScore. Het 10x van de functie aanroepen geeft de volgende resultaten:

```
Score goed tegen best is: 0
```

```
Score goed tegen best is: -1
```

Uit 10 keer de functie aanroepen volgde dat 5 potjes eindigde met een score van 0 en 5 potjes met een score van -1. Hieruit volgt dus een andere conclusie dan bij spel1: spel3 is tamelijk veel ingewikkelder dan spel1, en dus hoe ingewikkelder het spel, hoe “slechter“ bepaalGoedeZet zal werken en hoe meer besteScore zal winnen.

4 miljard standen

Om tot een miljard standen te komen leek het ons logisch om te kijken naar het verschil tussen spel1.txt, spel3.txt en spel4.txt. Ten eerste, spel3.txt heeft een stuk meer schalen dan spel1.txt en dit zorgde gelijk voor een gigantisch verschil in zowel performance als tijd. Maar zoals volgt uit spel4.txt het belangrijkste verschil zit niet alleen in het aantal schalen, maar juist in de combinatie van het aantal schalen en de pot. Als we bij spel1.txt ook maar twee karakters meer zouden toevoegen aan de pot - “gb“ - en er een schaal bij zouden doen kregen we terug:

```
Maak een keuze: 4

Beste score is: 0
Een beste zet is: (0,g)
We hebben hiervoor 24630 standen bekeken.
Dit kostte 16844 clock ticks, ofwel 0.016844 seconden.
```

Er waren dus bijna 3 keer zoveel clock ticks voor nodig!

Hieruit volgt dus dat wij voor ons 2.txt gekozen hebben voor een spel met een flinke pot (43 karakters) en veel schalen, 4 namelijk! Dit allemaal heeft ons geleid tot:

```
gbgbbbgbgbgggbbbbgggbgggbbggbgbgbgbgggbbgbb
4 5
7 6
1 0
4 0
0 5
0 0
7 0
0 0
8 0
3 0
0 4
0 2
0 7
0 0
4 0
5 0
0
```

Ofwel afgedrukt door de drukAf() functie:

```
Inhoud van de pot: bgggbbggbgbgbgbgggbbgbb
Inhoud van de schalen:
Schaal 0: g b g b b
Schaal 1: b g b g b
Schaal 2: g g g b b
Schaal 3: b b g g g
        Speler1     Speler2
Rij 1 :   1 0         3 0
Rij 2 :   4 0         0 4
Rij 3 :   0 5         0 2
Rij 4 :   0 0         0 7
Rij 5 :   7 0         0 0
Rij 6 :   0 0         4 0
Rij 7 :   8 0         5 0
Speler aan beurt: Speler 1
```

Tijdens het aanroepen van besteScore op spel2.txt gegeven dat spel2.txt het bovenstaande bevat was de uitvoer:

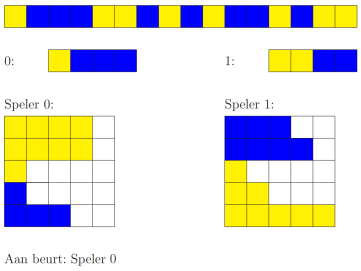
```
Maak een keuze: 4

Beste score is: 0
Een beste zet is: (0,g)
We hebben hiervoor 1573085709 standen bekeken.
Dit kostte 1485661574 clock ticks, ofwel 1485.66 seconden.
```

besteScore geeft ofwel dus aan dat de beste zet (0,g) is en dat als beide spelers optimaal spelen de eindscore 0 zal zijn. Ook staat in de cout hoeveel standen bekeken zijn en over hoeveel clock ticks dit gebeurde. Dit komt neer op 1486 seconden ofwel bijna 25 minuten over 1,5 miljard standen. Daarnaast is het ook wel leuk om het aantal bekeken standen per clock tick te berekenen. In dit geval was dat:

$$\frac{1573085709}{1485661574} = 1.058 \text{ standen bekeken per clock tick} \tag{3}$$

5 alternatief algoritme

Naast een optimaal algoritme gebruiken kan je ook tewerk gaan met algoritmes die niet het optimaal resultaat genereren. Een voorbeeld hiervan is het voorgestelde “gretige“ algoritme. Dit probeert elke zet zoveel mogelijk velden te vullen zonder uit te kijken naar de toekomst.



Een mogelijke positie als er verder wordt waar optimeel gespeeld is is als volgt:

```
Inhoud van de pot: gbgg
Inhoud van de schalen:
Schaal 0: g b b g
Schaal 1: g b b b
        Speler1     Speler2
Rij 1 :   4 0         0 5
Rij 2 :   4 0         0 4
Rij 3 :   3 0         1 0
Rij 4 :   0 4         5 0
Rij 5 :   0 5         5 0
Speler aan beurt: Speler 2
```

In deze positie zijn er twee mogelijke zetten, namelijk (0,g) en (1,g). De “gretige“ zet is (0,g), maar dit is de onjuiste zet, hiernaar zit speler1 in een voordeel en kan hij de positie forceren naar een win met +1. besteScore zou echter in de huidige positie kiezen voor de juiste zet: (1,g) en het spel zal eindigen met een score van 0:

```
Inhoud van de pot:
Inhoud van de schalen:
Schaal 0: b b b
Schaal 1:
        Speler1     Speler2
Rij 1 :   4 0         0 5
Rij 2 :   5 0         0 4
Rij 3 :   5 0         4 0
Rij 4 :   0 4         5 0
Rij 5 :   0 5         5 0
Speler aan beurt: Speler 1
```

```
Maak een keuze: 4

Beste score is: 0
Een beste zet is: (1,g)
We hebben hiervoor 13 standen bekeken.
Dit kostte 88 clock ticks, ofwel 8.8e-05 seconden.
```

Gebruikt spelbestand:

```
gbbbggbbgbbbggbgbgbbbgbgg\n
2 4\n5 5\n4 0\n4 0\n1 0\n0 1\n0 3\n0 3\n0 4\n1 0\n2 0\n5 0\n0
```

6 doeExperiment

doeExperiment is een functie die het spel tot het einde speelt op basis van zetten gemaakt door bepaalGoedeZet en vervolgens achteruit de berekeningstijd van besteScore meet. De functie maakt een kopie van het huidige spel (met behulp van *this op TegelSpel) en speelt dit uit totdat de eindstand is bereikt. Bij elke zet wordt het aantal zetten verhoogd.

Na het bereiken van de eindstand, gaat de functie elke zet terug en meet hoe lang het duurt voor besteScore om de huidige positie te berekenen. Dit wordt gedaan door de klok voor en na de aanroep van besteScore te meten en het verschil te berekenen. De berekende tijd wordt vervolgens naar de console geprint.

Als er meer dan één zet is gedaan, wordt ook het aantal zetten vanaf de startpositie geprint. Als er geen zetten meer over zijn, wordt de huidige stand van het spel afgedrukt.

De functie doeExperiment werkt dus door het spel vooruit te spelen tot het einde en vervolgens achteruit te gaan, waarbij bij elke stap de tijd wordt gemeten die nodig is om de beste score te berekenen. Dit geeft inzicht in hoe de berekeningstijd toeneemt naarmate er meer zetten ongedaan worden gemaakt en kan dus worden gebruikt om de efficiëntie van de besteScore functie te analyseren en te optimaliseren.

6.1 Resultaten

De resultaten van de functie voor de voorbeeldspellen in de tekstbestanden `spel4.txt` en `spel5.txt` zijn als volgt:

Spel4.txt		
Aantal Ongedaan Gemaakte Zetten	Aantal Bekeken Standen	Benodigde Tijd voor besteScore (s)
12	101.786.073	722.953
11	29.576.338	207.845
10	24.335.90	16.3639
9	10.648.40	7.06637
8	4.341.63	2.83764
7	668.45	0.438644
6	297.93	0.198507
5	158.87	0.107071
4	65.71	0.043244
3	71	0.000474
2	31	0.00024
1	0	1e-06

Spel5.txt		
Aantal Ongedaan Gemaakte Zetten	Aantal Bekeken Standen	Benodigde Tijd voor besteScore (s)
1	-	-
2	-	-
3	-	-
4	-	-
5	74.482.165	852.617
6	28.253.508	285.52
7	15.801.305	151.653
8	987.653	9.16108
9	124.028	1.15783
10	18.851	0.165502
11	6.599	0.060652
12	49	0.000402
13	0	1e-06

Op de plekken waar een streep staat deed het programma er dus te lang over er waren ofwel beëindigd.
Als we dit in een grafiek zetten zien we dit:

6.2 Conclusie

Als we kijken naar de resultaten van de bovenstaande diagrammen zien we bij elke tak die doeExperiment terug gaat de benodigde tijd voor het aanroepen van besteScore exponentieel vergroot. Dit komt doordat besteScore alle mogelijke zetten evalueert tot aan de maximale diepte. Voor diepere bomen is er een exponentieel toenemend aantal mogelijke zetten, wat de berekeningstijd enorm verhoogt.

6.3 Discussie

Het meest interessantste wat we opgemerkt hadden is dat het rechtstreeks aanroepen van besteScore zoals eerder gedaan is bij spel1 en spel3 10 keer zo sneller is, waarom dit precies zo is zijn we niet helemaal over uit. In eerste instantie lijkt het logisch: meer code ofwel dus meer tijd vereist. Maar hoe we de tijd meten staat rondom de aanroep van besteScore:

```
clock_t start = clock();
kopie.besteScore(besteZet, aantalStanden);
clock_t eind = clock();
```

dus dit zou eigenlijk geen verschil moeten maken lijkt ons. Dit fenomeen zien we bij elk spel dat we testen, waaronder ook spel1.txt en over de oorzaak kunnen we eigenlijk niks meer doen dan speculeren. Het enige logische verschil is dat de positie eerst benaderd wordt door bepaalGoedeZet, dus het zou kunnen dat bepaalGoedeZet kiest voor routen door de nodes heen die besteScore gelijk overslaat. Maar dit lijkt ons onlogisch aangezien we voor besteScore een vorm van het Minimax algoritme gebruiken.