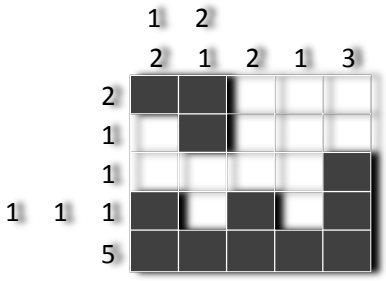
NONOGRAMMEN

OPLOSSEN MET DFS EN LOGISCHE REGELS

BACHELOREINDWERKSTUK KUNSTMATIGE INTELLIGENTIE

UNIVERSITEIT UTRECHT

JUNI 2013 [7,5 ECTS]



REMO VAN DER HEIDEN

BEGELEIDERS:
JEROEN GOUDSMIT
ROSALIE IEMHOFF

GEGEVENS

Student:	Remo van der Heiden
Opleiding:	Bachelor Kunstmatige Intelligentie, Universiteit Utrecht
Type eindwerkstuk:	Programmeeropdracht
Duur:	10 weken
1e beoordelaar:	Jeroen Goudsmit
2e beoordelaar:	Rosalie Iemhoff

VOORWOORD

Welkom in de wonderen wereld van de nonogrammen. De afgelopen tien weken heb ik me laten meeslepen in deze wereld, die vol verassingen zit. Vanaf het moment dat ik de eerste pagina van het onderzoek [1] had gelezen, heeft het me niet meer losgelaten. Ik wilde een programma schrijven wat nonogrammen kan oplossen. Ik begon met het programmeren van een “simpele”, maar wat later bleek, redelijk brute force zoekstrategie. Vervolgens probeerde ik een wat subtielere strategie met behulp van logische regels. Met veel plezier heb ik aan het programmeren en dit verslag gewerkt. Naast dat ik veel te weten ben gekomen oven nonogrammen, heb ik er veel van geleerd op het gebied van programmeren en datastructuren. Ik wil daarom ook graag mijn scriptiebegeleider Jeroen Goudsmit bedanken voor zijn ondersteuning en het voorstellen van dit onderwerp. Vooraf had ik nooit gedacht dat nonogrammen me zo zouden boeien.

Rotterdam, juni 2013

Remo van der Heiden

SAMENVATTING

Een nonogram is een logische puzzel waarbij door het combineren van gegeven cijfers, vakjes in een rooster kunnen worden ingevuld of leeggelaten. Op deze manier kan een plaatje worden onthuld. Omdat nonogrammen kunnen worden gezien als een zoekprobleem, is het niet makkelijk om er met een computer snel een oplossing voor te vinden. [1] stelt twee technieken voor die het mogelijk maken om met een computer de oplossing(en) van een nonogram te vinden. De eerste techniek is Depth First Search (DFS). Dit is een redelijk brute force zoekstrategie. De tweede techniek is subtieler, deze maakt gebruik van logische regels om de zoekruimte te verkleinen. Vervolgens wordt eventueel overgegaan op chronologisch backtracken, waarbij bij elke stap weer geprobeerd wordt om de zoekruimte opnieuw te verkleinen door middel van de logische regels.

Aan de hand van wat in [1] staat beschreven, is geprobeerd een herimplementatie te maken van zowel DFS, de logische regels en chronologisch backtracken. Op deze manier is gekeken of het mogelijk is om het onderzoek van [1] over te doen. Er is een raamwerk gemaakt wat ervoor zorgt dat puzzels kunnen worden ingelezen en een duidelijke digitale representatie kan worden weergegeven. Binnen dit raamwerk is het mogelijk oplossers te maken. Van beide technieken uit [1] is een oplosser gemaakt. De eerste oplosser gebruikt DFS om te zoeken door alle mogelijke oplossingen voor de rijen. Een oplossing wordt gevonden wanneer er een match is met de hints voor de kolommen. De tweede oplosser verkleint de zoekruimte stap voor stap, door het toepassen van logische regels. Het komt bij eenvoudige puzzels vaak voor dat het steeds proberen te verkleinen van de zoekruimte door middel van logische regels leidt tot het oplossen van de puzzel. Bij moeilijkere puzzels kunnen de logische regels de zoekruimte vaak wel verkleinen, maar wordt hij op een bepaald punt niet kleiner. [1] stelt hier het gebruik van chronologisch backtracken voor. Er is een aparte oplosser aangemaakt voor chronologisch backtracken. Deze oplosser vult elke keer één rij met een mogelijke oplossing en gebruikt de logische regels voor het valideren van deze rij en het proberen te verkleinen van de nieuwe zoekruimte. Beide strategieën zijn aan de hand van [1] te implementeren. Van sommige methodes is echter onbekend op welke manier ze geïmplementeerd zijn in [1]. Daardoor wordt het lastig om het onderzoek wat gedaan is precies te herhalen. Door het bedenken van eigen functies is het wel gelukt om een tweetal werkende oplossers te maken.

INHOUD

INLEIDING

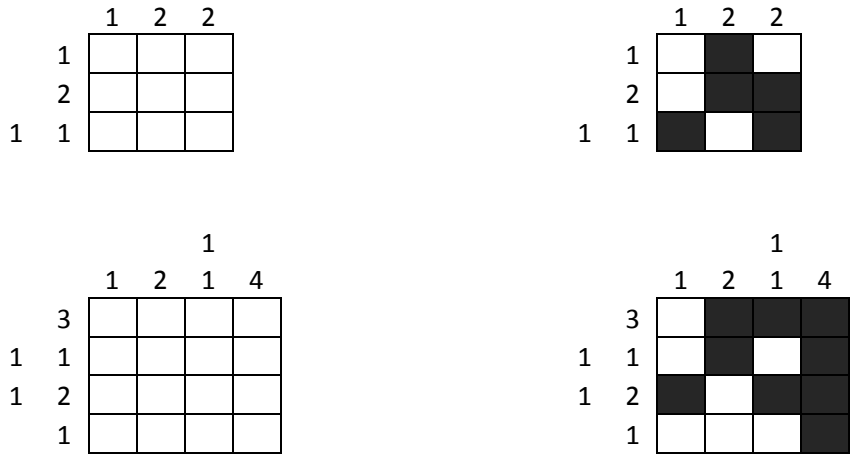
Een nonogram, ook wel Japanse puzzel genoemd, is een logische puzzel waarbij door het combineren van gegeven cijfers, vakjes in een rooster kunnen worden ingevuld of leeggelaten. Op deze manier kan een plaatje worden onthuld. In het volgende hoofdstuk wordt precies beschreven wat een nonogram is. Het met de hand oplossen van een nonogram is vaak een tijdrovend klusje. Dit komt doordat meestal niet in één oogopslag te zien is waar een vakje ingevuld dient te worden. Door het combineren van de gegeven informatie, is het vaak mogelijk om één of meerdere vakjes in te vullen of leeg te laten. Door elke keer op basis van de nieuw ingevulde of leeggelaten vakjes verder te redeneren, is het mogelijk stap voor stap de puzzel op te lossen.

Het oplossen van nonogrammen met behulp van een computer is een concreet voorbeeld van kunstmatige intelligentie. Het is voor de meeste puzzels namelijk niet mogelijk direct een oplossing te geven. Data dient vaak gecombineerd te worden. Een nonogram kan worden gekenmerkt als een zoekprobleem. [1] stelt een tweetal strategieën voor, voor het oplossen van nonogrammen. De eerste strategie is Depth First Search (DFS), dit is een redelijk brute force zoekstrategie die weinig intelligent te noemen is. De tweede strategie maakt gebruik van logische regels. Door het uitvoeren van deze logische regels op de puzzel, wordt geprobeerd de zoekruimte te verkleinen. Dit is een intelligente vorm van het (deels) oplossen van een zoekprobleem. [1] stelt voor de verkleinde zoekruimte vervolgens te doorzoeken met behulp van chronologisch backtracken. Dit is een vorm van DFS waarbij selectief wordt gezocht door paden die mogelijk tot een oplossing leiden. Interessant is om te kijken of het mogelijk is om de twee strategieën die [1] beschrijft aan de hand van dit artikel te herimplementeren. De onderzoeksvraag luidt dan ook: In hoeverre is het mogelijk om het zoeken van oplossingen van nonogrammen met behulp van Depth First Search en logische regels van [1] te herhalen. Wat zijn de haken en ogen aan deze aanpak?

Dit verslag beschrijft geeft eerst een korte introductie in nonogrammen. Hierna wordt het geen wat in code gemaakt is beschreven. Vervolgens wordt beschreven wat de resultaten zijn en geeft de conclusie weer wat verbeterd kan worden.

NONOGRAMMEN

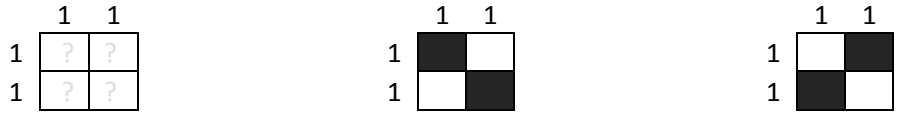
Een nonogram is een rechthoekige puzzel die gegeven wordt als twee rijen van rijen van cijfers (hints). De ene rij stelt de verticale hints voor, de andere de horizontale. De oplosruimte bestaat uit een rooster van vakjes (n x m). Deze vakjes kunnen drie waardes hebben: onbekend, leeg en ingevuld. De uitdaging is om uit de horizontale en verticale hints, in deze vakjes een plaatje te maken. De horizontale hints worden meestal boven de vakjes geplaatst, dit zijn de hints voor de kolommen. De verticale hints staan meestal links naast de vakjes, dit zijn de hints voor de rijen. Door het combineren van de hints kan worden bepaald welke vakjes ingevuld moeten worden. De overige vakjes blijven leeg. Een hint is een cijfer dat het aantal aaneengesloten in te vullen vakjes aangeeft. Dit noemen we een reeks. Tussen twee reeksen zit minimaal één leeg vakje. Als een rij bijvoorbeeld twee hints heeft: 1 2, dan moet er ergens in deze rij één vakje ingevuld worden, vervolgens worden na minimaal één vakje leeg gelaten te hebben twee vakjes naast elkaar ingevuld. Zie de derde rij van het tweede nonogram van figuur 1.



Figuur 1. Twee nonogram en bijbehorende unieke oplossing.

OPLOSSINGEN

Een oplossing voor een nonogram is een invulling van de vakjes die overeenkomt met de hints. Een nonogram kan één of meerdere oplossingen hebben. In figuur 2 wordt een voorbeeld gegeven van een nonogram met meerdere oplossingen.



Figuur 2. Een nonogram en zijn twee oplossingen

Bij een nonogram wordt er vanuit gegaan dat ieder in te vullen vakje in zowel de horizontale als de verticale hints wordt gerepresenteerd. Het totaal aantal te vullen vakjes komt overeen met de som van horizontale hints en de som van de verticale hints. Als dit niet het geval is, is de puzzel onoplosbaar.

COMPLEXITEIT

Het is bekend dat het probleem van het vinden van een oplossing van een nonogram NP volledig is [2]. Dit betekent dat nonogrammen niet in polynomiale tijd oplosbaar zijn als P niet gelijk is aan NP.

RAAMWERK

Om te beginnen is er een raamwerk gemaakt. Dit maakt het mogelijk om te werken met nonogrammen in Java. Het raamwerk bestaat uit een klasse die de puzzel representeert. In deze klasse wordt de puzzel opgeslagen als een tweedimensionale array die de vakjes van de puzzel representeert. Verder zijn er functies gemaakt om een puzzel in te lezen, te transformeren en af te drukken op het scherm. De klasse Solver is gemaakt om te dienen als basis voor een oplosser voor nonogrammen. Solver bevat een functie om een rij of kolom van een puzzel te permuteren.

We nemen de belangrijkste onderdelen van het raamwerk onder de loep.

KLASSE PUZZLE

VARIABLEN

Naam	Type	Gegevens	Functie
hintsH	*int[][]*	*[kolom][hint]*	Opslaan invoer horizontaal
hintsV	*Int[][]*	*[rij][hint]*	Opslaan invoer verticaal
puzzle	*char[][]*	*[rij][kolom]*	Representatie van de puzzel
transposedpuzzle	*char[][]*	*[kolom][rij]*	Representatie van de puzzel
savestream	*Stack<char[][]>*	*puzzle:char[][]*	Opslaan van puzzel

FUNCTIES

int[][] splitpuzzle(String input)
De functie splitpuzzle verwacht als input een *String* met regels van hints gescheiden door komma's. Deze hints worden per regel in *int[][]* gestopt.

void generateGrid()
De functie generategrid maakt een nieuwe lege *puzzle:char[][]* en *transposedpuzzle:char[][]* aan.

String topuzzle()
De functie topuzzle genereert een *String* waarbij de horizontale en verticale hints worden gecombineerd met de huidige (oplossing van) *puzzle:char[][]*.

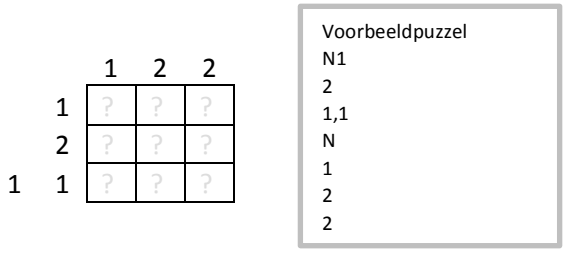
boolean complete()
De functie complete is true als er geen onbekende vakjes meer in de puzzel voorkomen. De functie complete zegt niets over de validiteit van de puzzel.

void save(), void restore(), void remove()
Deze functies zorgen ervoor dat de puzzel kan worden opgeslagen in een stack. De functie *void save()* voegt een puzzel toe aan de stack (push). De functie *void restore()* haalt de laatst opgeslagen puzzel terug (peek). De laatst opgeslagen puzzel wordt verwijderd door *void remove()* (pop).

BESCHRIJVING

De klasse Puzzle kan worden aangemaakt. Deze heeft als parameter een puzzelnummer*:int*, het nummer van de te laden puzzel. De constructor van Puzzle zorgt ervoor dat na het aanmaken van een *<Puzzle>* de puzzel wordt geladen. Alle informatie die nodig is voor het oplossen van de puzzel staat nu klaar in de variabelen.
De invoer van een puzzel gebeurt middels een tekstbestand. Deze bestanden hebben de extensie *.pzl, verwijzend naar puzzel. Zo'n bestand bevat de hints van de puzzel. Als eerste de verticale hints, die naast de puzzel staan (hints voor de rijen). Vervolgens de horizontale hints, die boven de puzzel staan (hints voor de kolommen). Om de gegevens op te splitsen wordt het karakter 'N' gebruikt. De afzonderlijke hints worden gescheiden door een komma en de verschillende rijen hints door een line feed ('\n'). Voor de gegevens van de puzzel is ruimte voor notities, hier kan bijvoorbeeld de naam of

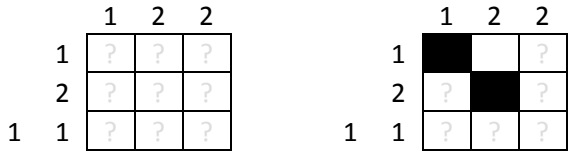
het aantal oplossingen worden vermeld. De notitieruimte wordt ook gesplitst van de hints met een line feed. Als voorbeeld, de inhoud van een *.pzl bestand:
“Voorbeeldpuzzel\nN1\n2\n1,1\nN1\n2\n2\n”. Zie figuur 3.



Figuur 3. Een nonogram en zijn *.pzl bestand.

De gegevens worden opgedeeld door te splitsen op ‘N’. Er blijven nu drie *Strings* over. Met de eerste *String* doen we niets. De tweede *String* zijn de horizontale hints, de derde *String* de verticale. We voeren deze *Strings* beide in, in de functie splitpuzzle en krijgen hintsH en hintsV als output. Nu kunnen we een puzzle en transposedpuzzle aanmaken. Beide zijn een representatie van de vakjes van de puzzel. Bij transposedpuzzle worden de rijen en kolommen gewisseld van plek. Dit zorgt ervoor dat kolommen als rijen kunnen worden doorlopen. Het nadeel van een tweedimensionaal *char* array is namelijk dat een kolom niet makkelijk te doorlopen is.

Er wordt van uitgegaan dat de invoer altijd een lege puzzel is, waarbij alle vakjes nog onbekend zijn. In puzzle en transposedpuzzle maken we onderscheid tussen drie typen: onbekend, ingevuld en leeg. Deze krijgen in puzzle en transposedpuzzle een *char* als representatie. Het lege karakter (‘\n’) representeert een onbekend vakje. ‘1’ en ‘0’ representeren respectievelijk ingevuld en leeg. De linker puzzel uit figuur 4 wordt gerepresenteerd door de volgende puzzle *char[][]*: { {‘’,’’,’’} , {‘’,’’,’’} , {‘’,’’,’’} } , transposedpuzzle heeft in dit geval de zelfde representatie. De rechter puzzel wordt in puzzle gerepresenteerd als: { {‘1’,’0’,’’} , {‘’,’1’,’’} , {‘’,’’,’’} }. In dit geval heeft transposedpuzzle een andere representatie: { {‘1’,’’,’’} , {‘1’,’0’,’’} , {‘’,’’,’’} }.



Figuur 4. Een puzzel met twee verschillende invullingen.

De functie topuzzle zet de puzzel om in een *String* zodat deze op het scherm weergegeven kan worden. Voor elk vakje wat gevuld is wordt een “#” geprint. Voor onbekende vakjes wordt een “?” geprint en voor lege vakjes een “ “. Boven en voor de puzzel worden de hints toegevoegd. Zie figuur 5.

```
    1 2 2
  1 ? ? ?
  2 ? ? ?
1 1 ? ? ?
```

```
    1 2 2
  1 # ? 
  2 ? # ?
1 1 ? ? ?
```

Figuur 5. De interne weergave van de puzzels uit figuur 4.

KLASSE SOLVER

Deze klasse geeft de basis voor een oplosser. DFS en Rulebased extenden Solver en kunnen op die manier gebruik maken van gezamenlijke functies en variabelen die nodig zijn voor het oplossen van nonogrammen.

Solver bevat de volgende variabelen:

Naam	Type	Gegevens	Functie
empty	*final static char*	*'0'*	Representatie leeg vakje
colored	*final static char*	*'1'*	Representatie ingevuld vakje
unknown	*final static char*	*'\0'*	Representatie onbekend vakje
puzzlesizeH	*int*		Puzzelgrootte horizontaal
puzzlesizeV	*int*		Puzzelgrootte verticaal
puzzle	*char[][]*	*[rij][kolom]*	Representatie van de puzzel
transposedpuzzle	*char[][]*	*[kolom][rij]*	Representatie van de puzzel

Solver bevat drie statische karakters die er voor zorgen dat de code overzichtelijk blijft. De waardes van puzzle en transposedpuzzle kunnen op deze manier gebruikt worden als static variable in de code.

Solver bevat de volgende functie:
Set<String> permute(String line, int zeros, Set<String> solution)

De functie permute heeft als invoer drie argumenten:

- *String line*, de te permuteren rij;
- *int zeros*, het aantal lege cellen aan het eind;
- *Set<String> solution*, de verzameling waar de gevonden permutaties in worden opgeslagen.

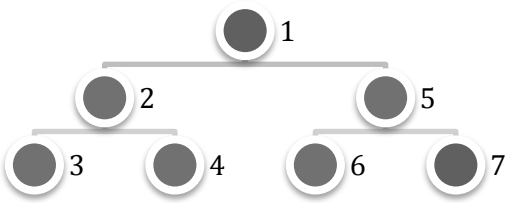
Permute is recursief. Als een gevonden permutatie nog lege cellen aan het eind heeft, wordt permute aangeroepen. Op deze manier worden alle permutaties die voldoen aan de hints gevonden.

BESCHRIJVING

De precieze werking van permute wordt beschreven bij de implementatie van Depth First Search.

DEPTH FIRST SEARCH (DFS)

De eerste aanpak die we onder de loep hebben genomen, is die met een Depth First Search algoritme (DFS). DFS is een zoekalgoritme dat gebruik maakt van een zoekboom. Een zoekboom bestaat uit knopen. Bij DFS wordt, zodra een knoop bezocht is, eerst gezocht in de kinderen van deze knoop. Als de knoop geen kinderen heeft, wordt teruggegaan naar de vorige knoop en wordt het volgende kind daarvan bezocht. Voor het zoeken naar de oplossing van een nonogram wordt voor elke rij alle mogelijke invullingen uitgerekend. Dit noemen we ook wel mogelijke permutaties. Als we deze permutaties combineren met alle mogelijke permutaties van de andere rijen krijgen we alle mogelijke permutaties van de puzzel. De lagen van de zoekboom stellen de rijen van de puzzel voor. De bovenste laag knopen bevat de verschillende permutaties van de eerste rij. De tweede laag knopen bevat de verschillende permutaties van de tweede rij, enzovoort. In figuur 6 is een zoekboom gegeven. Een zoekboom zoals deze zou ontstaan bij een puzzel van drie rijen. Deze zou op de eerste en tweede rij maar één mogelijke oplossing hebben, op de derde rij twee.



Figuur 6. Een zoekboom met de volgorde van DFS aangegeven.

COMPLEXITEIT

Om een idee te krijgen over de complexiteit van nonogrammen, proberen we ze eerst op te lossen met het DFS algoritme. Voor elke rij worden alle mogelijke permutaties gegenereerd. Als we kijken naar de hoeveelheid permutaties bij een kleine puzzel dan lijdt dit tot een paar per rij. Dit levert al snel een hoop permutaties voor de totale puzzel op.

	Lengte van de rij						
		4	**5**	**6**	**7**	**8**	**9**	**10**
hints	**1**	4	5	6	7	8	9	10
	11	3	6	10	15	21	28	36
	111	X	1	4	10	20	35	56
	21	1	3	6	10	15	21	28
	2	3	4	5	6	7	8	9

Tabel 1. Het aantal permutaties voor verschillende combinaties van rijlengte en hints.

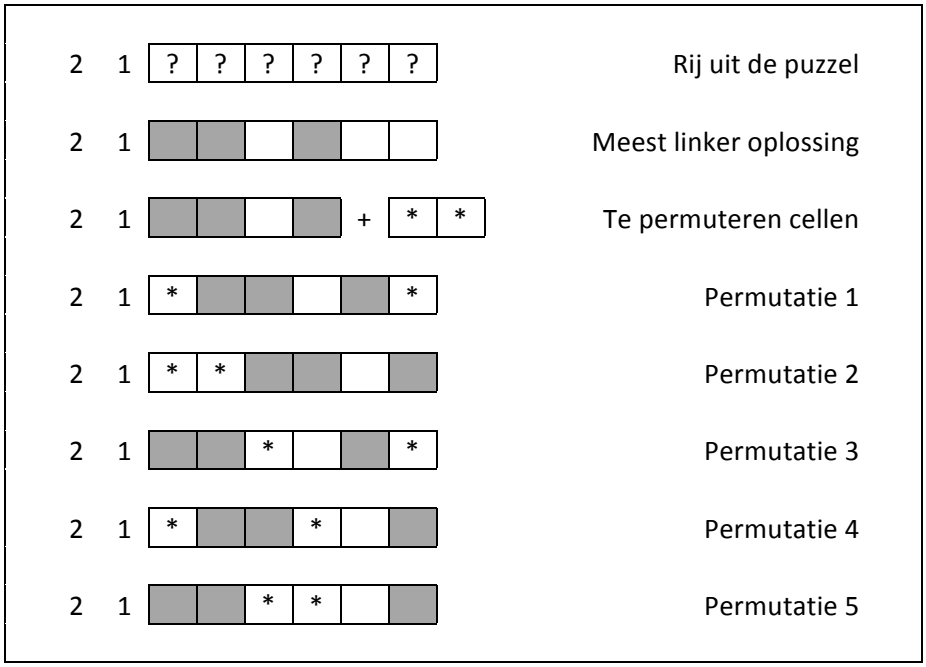
Als we kijken naar tabel 1 kunnen we concluderen dat er een grote differentiatie zit in de hoeveelheid mogelijke permutaties. Als (totale lengte van hints * 2) – 1 dicht bij de lengte van de rij ligt, of het aantal hints laag is, worden weinig permutaties gegenereerd. Dit betekent in de praktijk voor deze aanpak, dat het langer duurt om een puzzel op te lossen die rond het midden van deze uitersten zit. Dit omdat er in dat geval veel permutaties worden gegenereerd, waar doorheen gezocht moet worden.

IMPLEMENTATIE

Om DFS te kunnen implementeren hebben we een zoekboom nodig om doorheen te zoeken. We bouwen deze zoekboom door alle mogelijke combinaties van rijen te maken. Als een combinatie van mogelijke rijen matcht met de hints voor de kolommen, hebben we een oplossing voor de puzzel gevonden.

HET GENEREREN VAN DE ZOEKBOOM

Om alle mogelijke oplossingen te genereren, genereren we per rij alle mogelijke oplossingen. Als we van elke rij één zo'n oplossing nemen vormt dit een mogelijke oplossing voor de puzzel. Om alle mogelijke oplossingen van een rij te genereren bouwen we de meest linker oplossing op. De informatie voor deze rij komt uit de verticale hints. We hebben nu de eerste mogelijke oplossing voor deze rij. Vervolgens permuteren we de nullen die rechts over blijven voor de secties met gevulde vakjes (figuur 7).

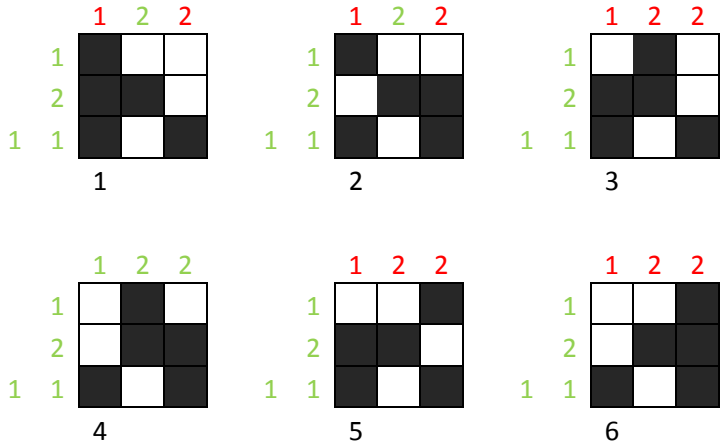


Figuur 7. Het genereren van de meest linker oplossing en het permuteren ervan.

Voor elke rij herhalen we dit proces. We slaan de meest linker oplossing en alle permutaties per rij op in een array die we later kunnen doorzoeken.

HET ZOEKEN

Een optie zou zijn om een daadwerkelijke zoekboom op te bouwen. In dit geval is dit niet nodig, omdat door het combineren van de data van de rijen al door de oplossingen gezocht kan worden. Dit werkt als volgt. Er wordt een eerste oplossing gemaakt, door van elke permutatie-array het eerste element te nemen. We hebben nu van elke rij de meest linker oplossing. Vervolgens kijken we of dit een oplossing voor de puzzel is, door te kijken of de puzzel voldoet aan de horizontale hints, die aangeven wat er in de kolommen moet staan. Aangezien een puzzel meerdere oplossingen kan hebben, zoeken we verder totdat we alle mogelijkheden getoetst hebben. Op deze manier worden altijd alle oplossingen gevonden.



Figuur 8. DFS zoeken naar de oplossing (4).

KLASSE DFS

De klasse DFS extend de klasse solver met de functies die nodig zijn om alle oplossingen van een nonogram te vinden via DFS. Er zijn vier functies waar DFS gebruik van maakt, “permutations”, “permute”, “searchsolution” en “checksolution”.

FUNCTIES

DFS gebruikt de volgende variabelen:

Naam	Type	Gegevens	Functie
possibilities	*Iterator<String>[]*	*[permutatie]*	Itereren door permutaties mogelijk maken
line	*Int*		Huidige rij aangeven
permutations	*ArrayList<Set<String>>*	*[rij][permutatie]*	Lijst met lijst van permutaties
solution	*char[][]*	*[rij][kolom]*	Representatie van de puzzel
solutionhints	*String*		Doorgeven voor oplossing controle
pu	*<Puzzle>*		Doorgeven puzzel

DFS gebruikt de volgend functies:

Set<String> permutations(int[] line,int size)
Deze functie genereert de meest linker oplossing van een rij. Deze oplossing wordt ingevoerd in permute, die alle mogelijke permutaties teruggeeft.

Set<String> permute(String line, int zeros, Set<String> solution)
Deze functie permuteert de nullen op het eind van een oplossing op alle manieren tussen de reeksen.

Boolean checksolution(char[][] solution,String originalhintstring, Puzzle pu)
Deze functie test of een ingevoerde oplossing ook voldoet aan de hints van de kolommen.

void searchsolution(Iterator<String>[] possibilities, int line, ArrayList<Set<String>> permutations,char[][] solution,String solutionhints,Puzzle pu)
De functie searchsolution zoekt recursief naar oplossingen voor een puzzel.

Het doel van DFS is om alle oplossingen van een puzzel te vinden. Als start krijgt DFS een *<Puzzle>* mee. Hierin zit alle informatie die nodig is om een puzzel op te lossen. De eerste stap is het genereren van de meest linker oplossing van elke rij. Dit wordt gedaan door te beginnen met de eerste hint van de eerste rij. Plak een "1" aan de *String* en verlaag de hint met één. Als de hint nul is geworden, plak een "0" aan de *String* en ga naar de volgende hint. Ga zo door tot de laatste hint en plak daar geen "0" achter.

```
while hints is not empty
  while hints[i] is not 0
     add "1" to output
     hint[i] minus 1
add "0" to output if not last hint
```

Figuur 9. Pseudocode voor het genereren van de meest linker oplossing

Het verschil in grootte tussen de lengte van de rij en de gemaakte meest linker oplossing (zonder laatste nullen), is het aantal nullen (ofwel lege vlakjes) dat tussen de reeksen gepermuteerd kan worden. Merk op dat als er geen verschil is in lengte tussen de meest linker oplossing en de lengte van rij, er maar één oplossing voor deze rij mogelijk is.

Het permuteren gebeurt in permute. Deze functie krijgt mee hoeveel nullen (zeros) er nog aan het eind staan om te permuteren. Begonnen wordt met het plaatsen van de eerste nul voor elke reeks. Voor elke plaatsing wordt opnieuw permute aangeroepen met zeros -1. Dit gaat zo verder tot er geen nullen meer zijn om te permuteren. Figuur 10 illustreert het permuteren. Dit levert de volgende *String*s op: "110100", "011010", "001011", "110010", "011001", "110001". Duidelijk is te zien dat de laatste nullen worden verdeeld tussen de reeksen enen. Twee nullen verdelen over drie plekken geeft 2x3 mogelijkheden.

```
if zeros is 0
  add to set
else
   for every character of the row
      if not at last character
         if (at first character and this is 1)
         or (character is 0 and next is 1)
            insert 0 and cut 0 from end
            run recursive with zeros -1
      else if last character
         run recursive with zeros -1
```

Figuur 10. Pseudocode voor het genereren van de permutaties

Nu de permutaties van alle verschillende rijen zijn gegenereerd, is door het maken van alle combinaties het zoeken volgens DFS mogelijk. Hiervoor worden de *String*s omgezet in *char[]*, zodat het combineren van de rijen een totale puzzel oplevert. De functie searchsolution combineert alle permutaties van rij één met alle permutaties van rij twee. Rij twee combineert zichzelf met alle combinaties van rij drie, enzovoort. Nadat een combinatie van alle rijen is ontstaan wordt deze ingevoerd in checksolution.

```
if no more permutations in this row
  go to next row
  while possibilities in this row
    check current solution
    load next permutation in
    run recursive with row + 1
```

Figuur 11. Pseudocode voor searchsolution

De laatste stap is het controleren van de oplossing. Hiervoor worden de horizontale hints opgebouwd uit de puzzel. Wanneer de hints matchen met de hints van de puzzel is er een oplossing voor de puzzel gevonden. Voor elke reeks in een kolom wordt de lengte een hint. De hints worden gescheiden met komma's en de kolommen met een '\n' (line feed).

```
for each column
  for each sequence
     put length of sequence
     put "," if not last sequence
  put "/n"

if generated String equals hintstring
      return true
```

Figuur 12. Pseudocode van checksolution.

Als er een oplossing gevonden is, dient er verder gezocht te worden. Zoals bekend kunnen er meerdere oplossingen zijn. Op deze manier worden ze allemaal gevonden.

LOGISCHE REGELS

Deze aanpak maakt gebruik van een set logische regels. Een logische regel voert een actie uit als iets voldoet aan bepaalde eisen. In dit geval wordt naar een rij of kolom uit de puzzel gekeken. Als actie worden vakjes gevuld of leeg gelaten. Een voorbeeld van een logische regel die van toepassing is op nonogrammen, is de volgende. Als deze rij maar één hint heeft en deze is gelijk aan de lengte van de rij, vul alle vakjes van de rij in.

Figuur 13. Voorbeeld van een logische regel.

[1] heeft een aantal al bekende regels uit eerdere onderzoeken gecombineerd met nieuwe regels, waardoor het mogelijk wordt de meeste simpele puzzels (puzzels met één enkele oplossing) snel op te lossen. Het idee van Rulebased is het stap voor stap door de puzzel heen lopen. Gestart wordt bij de eerste rij. De logische regels worden na elkaar op deze rij toegepast. Dit wordt herhaald tot de laatste rij. Vervolgens wordt het proces herhaald voor de kolommen. Als de puzzel nog niet opgelost is, herhaal dan het hele proces. De aanpak met logische regels heeft veel weg van de aanpak van de mens voor het oplossen van nonogrammen. Veel mensen lossen nonogrammen namelijk ook op door stap voor stap de hints te bekijken en stap voor stap de informatie te verwerken in de puzzel.

In sommige gevallen is het achtereenvolgens blijven toepassen van de logische regels op de puzzel niet voldoende om een valide oplossing voor de puzzel te vinden. De oplossing hiervoor is het maken van een gok en kijken hoe de uitwerking hiervan op de puzzel is. [1] stelt voor om alle mogelijke oplossingen van een rij te genereren. Deze kunnen dan stuk voor stuk worden ingevuld in de puzzel en vervolgens kunnen de logische regels er op losgelaten worden.

```
while puzzle not solved
  loop over rows
    loop over rules
  loop over columns
    loop over rules
```

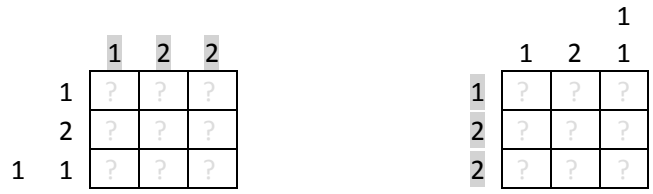
Figuur 14. Pseudocode van rulebased

COMPLEXITEIT

Uit het idee van de aanpak blijkt al dat ook deze methode een hoop rekenwerk vergt. Op het eerste gezicht lijkt deze aanpak in ieder geval een stuk subtieler. Het blijkt echter zo te zijn dat het niet altijd mogelijk is om via de logische regels tot een oplossing te komen. Zelfs niet voor puzzels met maar één enkele oplossing. Interessant hierbij is de vraag of het oplossen van nonogrammen met één unieke oplossing ook een NP volledig probleem is.

IMPLEMENTATIE

Er is geprobeerd de voorgestelde en uitgewerkte methode uit [1] te implementeren. In sommige gevallen zijn er aannames gemaakt. [1] is niet zo precies over hoe moet worden omgegaan met randgevallen. Om vast te stellen en bij te houden waar mogelijk vakjes kunnen worden ingevuld, wordt het bereik van de hints vastgesteld en bijgehouden. Het bereik van een hint wordt in [1] ook wel een “blackrun” genoemd. [1] stelt een elftal logische regels voor, die zich richten op één enkele rij of kolom. Deze regels kunnen we onderverdelen in drie categorieën. Regels die vakjes vullen of leeglaten, regels die het bereik van een hint aanpassen en regels die beide combineren. Door het achter elkaar uitvoeren van alle regels op alle rijen en vervolgens op alle kolommen, wordt geprobeerd een oplossing te vinden. Om het doorzoeken van een kolom mogelijk te maken, wordt de puzzel getransponeerd, waardoor de kolommen rijen worden en andersom. Op deze manier is het mogelijk om als input voor de regels altijd rij-informatie te geven. In figuur 15 wordt dit uitgebeeld. [1] stelt wel dat rijen het zelfde zien als kolommen, maar zegt niet op welke manier er gezocht wordt door rijen en kolommen.



Figuur 15. Transponeren.

LOGISCHE REGELS (KLASSE RULEBASED)

KLASSE BLACKRUNS

Om per rij en per kolom het bereik van iedere hint op te slaan, is er een klasse “blackruns” aangemaakt. Deze klasse bevat de voor elke hint het bereik en andere rij- en kolomrelevante informatie

Blackruns bevat de volgende variabelen:

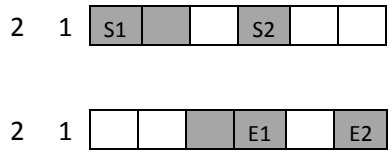
Naam	Type	Gegevens	Functie
js	*int[]*	*[hint]*	Aangeven startpositie bereik
je	*int[]*	*[hint]*	Aangeven eindpositie bereik
k	*int*		Aantal blackruns (= aantal hints)
n	*int*		Lengte van de rij/kolom
hint	*int[]*	*[hint]*	Hint die het bereik hoort
line	*char[]*	*[vakje]*	Rij/kolom van de puzzel
solved	*boolean*		Opgelost of niet

Verder bevat Blackruns een constructor die een deel van de bovenstaande variabelen initieert. line blijft in eerste instantie leeg, deze worden later gevuld en gebruikt.

LOGISCHE REGELS

Blackruns initialrunrangeestimation(char[] line, int[]hints)
Deze functie zorgt ervoor dat per rij en kolom een *<Blackruns>* wordt aangemaakt en vult deze met de juiste informatie. Voor elke rij wordt de meest linker met de meest rechter oplossing gecombineerd. Hierdoor wordt duidelijk wat de ranges van de verschillende te plaatsen gevulde vlakjes zijn. De meest linker oplossing wordt verkregen door voor elke hint het aantal vlakjes te

combineren, dit vervolgens te combineren met de andere hints en tussen iedere hint een vlakje leeg te laten. De meest rechter oplossing krijgen we door vanaf het eind van de rij/kolom te beginnen en in tegengestelde richting de vlakjes per hint te plaatsen. Ook hier houden we natuurlijk één vlakje leeg na iedere hint.



Figuur 16. Meest linker en meest rechter oplossing van een rij.

In figuur 16 zien we de meest linker en meest rechter oplossing van een rij. Het bereik van de hints kunnen nu als volgt worden bepaald. Voor het startpunt van het bereik nemen we de plek van het eerste gevulde vakje uit de meest linker oplossing wat bij deze hint hoort. Voor het eindpunt gebruiken we de plek van het laatste gevulde vakje uit de meest rechter oplossing. Het bereik wat hoort bij de eerste hint (2) is (S1,E1), dus (0,3). Het bereik wat hoort bij de tweede hint (S2,E2), is (3,5). In [1] begint met het nummeren van blackruns bij 1 in plaats van 0 wat voor het programmeren een stuk logischer zou zijn. Dit is inconsistent met een rij uit de puzzel, deze wordt wel vanaf 0 genummerd.

De functie initialrunrangeestimation heeft twee parameters. De rij (gerepresenteerd als *char[]*) en de hints (gerepresenteerd als *int[]*). Het uitvoeren van deze functie levert een *<Blackruns>* op.

REGELS DEEL 1

Deel 1 bevat vijf regels die zorgen voor het vullen of leeglaten van de vakjes op basis van het bereik van de hints. Als bijvoorbeeld de lengte van het bereik gelijk is aan de hint, hebben we een match en kunnen we de vakjes voor deze hint direct vullen.

Ieder van deze vijf regels heeft de volgende vorm:

char[] Regel*(Blackruns R, char[]line)*

Er worden twee argumenten meegegeven, *<Blackruns>* en line. line is een *char[]* die de huidige rij voorstelt. De output van deze functie is die zelfde line, maar dan met de regel er op toegepast.

Regel 1:

Een vakje wordt ingevuld wanneer binnen het bereik van een hint, de meest linker en meest rechter oplossing overlappen.

Figuur 17. De meest linker en meest rechter oplossing overlappen.

In figuur 17 zien we hoe regel 1 werkt op een rij. Het bereik van de hint “3” is in dit geval (0,4). In figuur 16 wordt een rij weergegeven waarbij geen overlap is binnen het bereik van de hints. In het geval van figuur 16 zou dus geen vakje gevuld zijn.

Regel 1.1 is in het project geïmplementeerd als: *char[] Ione(Blackruns R, char[]line)*

Regel 2:
Voor elk vakje geldt: als het niet behoort tot het bereik van een hint, dan moet het leeg zijn. Deze regel kan weer worden opgedeeld in drie sub regels:

- Alle vakjes voor het bereik van de eerste hint kunnen worden leeggelaten;
- Alle vakjes na het bereik van de laatste hint kunnen worden leeggelaten;
- Voor iedere twee opeenvolgende bereiken, kunnen we alle ertussen liggende vakjes leeglaten.

Figuur 18. Alle vakjes na het bereik van de laatste hint kunnen worden leeggelaten

Het toepassen van deze regel heeft alleen nut als het bereik van één of meerdere hints is aangepast. In eerste instantie is het bereik van alle hints samen gelijk aan de totale rij. Het uitvoeren van deze regel tijdens de eerste iteratie heeft dus geen toegevoegde waarde.

Regel 1.2 is in het project geïmplementeerd als: *char[] Itwo(Blackruns R, char[]line)*

Regel 3:
Als voor het bereik van elke hint geldt:

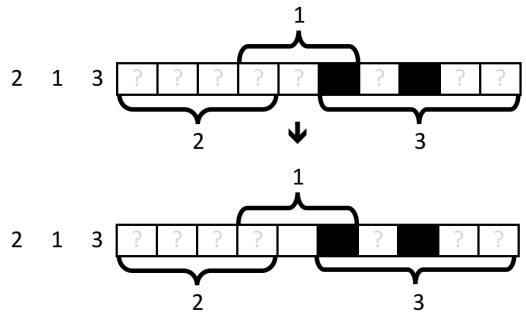
- Het eerste vakje is ingevuld;
- De rest van de hints die dit vakje in het bereik heeft is “1”.

Laat het vakje voor dit bereik leeg.

De andere kant op, als voor het bereik van elke hint geldt:

- Het laatste vakje is ingevuld;
- De rest van de hints die dit vakje in het bereik heeft is “1”.

Laat het vakje na dit bereik leeg.

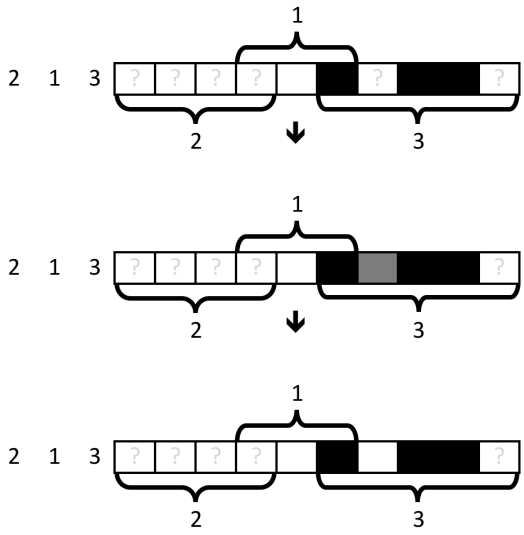


Figuur 19. Een rij voor en na het toepassen van regel 1.3, bereik per hint aangegeven.

Regel 1.3 is in het project geïmplementeerd als: *char[] Ithree(Blackruns R, char[]line)*

Regel 4:
Als binnen het bereik van een hint geldt:
Als twee reeksen van ingevulde vakjes gescheiden worden door een onbekend vakje en we zouden door het vullen van dit vakje een reeks krijgen die langer is dan de grootste bijbehorende hint, laat dit vakje leeg.

Gezien het bereik van verschillende hints kan overlappen, kunnen meerdere hints van toepassing zijn op een vakje. Deze regel geldt dus alleen als alle hints waarvan het bereik het vakje bevat, kleiner zijn dan de ontstane reeks.



Figuur 20. Een rij voor en na het toepassen van regel 1.4, bereik per hint aangegeven.

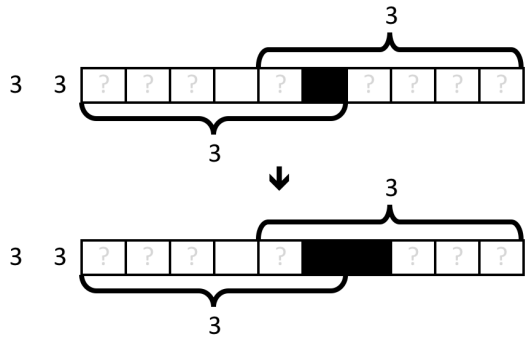
Regel 1.4 is in het project geïmplementeerd als: *char[] Ifour(Blackruns R, char[]line)*

Regel 5:
Als een leeg of onbekend en een ingevuld vakje samen voorkomen in het bereik van een hint, kunnen we soms extra vakjes vullen:

Voor elk vakje *i* uit deze rij:

- Stel dat het vakje ingevuld is en het vakje ervoor leeg of onbekend, laat dan minL de minimale lengte zijn van de hints waarvan het bereik het vakje bevat.
- Stel dat zich links van vakje *i,* binnen (minL +1) afstand, een leeg vakje *j* bevind. Vul dan elk vakje rechts van vakje *i* in dat kleiner is dan vakje *j* + minL.
- Stel dat zich rechts van vakje *i,* binnen (minL -1) afstand, een leeg vakje *k* bevind. Vul dan elk vakje links van vakje *i* in dat groter is dan vakje *k* - minL.

- Als alle hints waarvan het bereik het vakje bevat gelijk zijn aan de lengte van de reeks. Laat dan voor en na de reeks een vakje leeg.



Figuur 21. Een rij voor en na het toepassen van regel 1.5, bereik per hint aangegeven.

Het laatste deel van de regel kunnen we zien als een aparte regel, deze kan los worden toegepast op een rij.

Regel 1.5 is in het project geïmplementeerd als: *char[] Ifive(Blackruns R, char[]line)*

REGELS DEEL 2

Deel 2 houdt zich bezig met het aanpassen van het bereik van de hints. In dit deel wordt in de puzzel niets ingevuld. Dit deel bestaat uit drie regels.

Ieder van deze drie regels heeft de volgende vorm:

Blackruns Regel*(Blackruns R, char[]line)*

Er worden twee argumenten meegegeven, *<blackruns>* en line. De *char[]* line stelt de huidige rij voor. De output van deze functie is diezelfde *<blackruns>*, maar dan met de regel erop toegepast.

Regel 1:
Als het startpunt van het bereik van een hint voor of op het startpunt van het bereik van de vorige hint ligt, pas het startpunt als volgt aan. Startpunt = (startpunt bereik vorige hint) + (vorige hint) +1. Op die manier genereren we de minimale ruimte die nodig is voor het vullen van de vakjes van de vorige hint.

Ditzelfde geldt ook voor het eindpunt van het bereik van een hint. Als het eindpunt van het bereik van een hint na of op het eindpunt van het bereik van de volgende hint ligt, pas het eindpunt als volgt aan. Eindpunt = (eindpunt bereik volgende hint) – (volgende hint) -1.

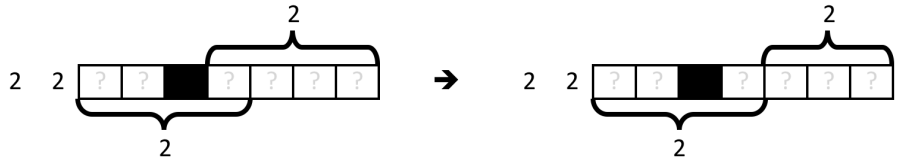
Noot: Er dient altijd één vakje extra ruimte gehouden te worden. Dit is voor het lege vakje tussen de reeksen.

Regel 2.1 is in het project geïmplementeerd als: *Blackruns IIone(Blackruns R, char[]line)*

Regel 2:
Voor het bereik van elke hint:

Als het vakje wat direct voor het bereik van de hint zit, ingevuld is. Maak het bereik dan kleiner door het één vakje verder naar rechts te laten starten.

Als het vakje wat direct na het bereik van de hint zit, ingevuld is. Maak het bereik dan kleiner door het één vakje verder naar links te laten starten.



Figuur 22. Een rij voor en na het toepassen van regel 2.2, bereik per hint aangegeven.

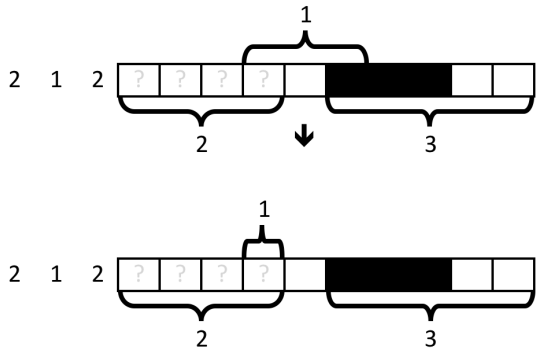
Deze regel zorgt er dus voor dat er rekening gehouden wordt met het lege vakje wat tussen iedere reeks zit.

Regel 2.2 is in het project geïmplementeerd als: *Blackruns IItwo(Blackruns R, char[]line)*

Regel 3:
Als een reeks ingevulde vakjes groter is dan het bereik van een hint:

Kijk of deze reeks binnen het bereik van een van de volgende hints past en niet groter is dan deze hint. Als dit zo is, pas het bereik aan zodat dit twee vakjes voor de reeks eindigt.

Als deze reeks niet bij het bereik van een van de volgende hints hoort, kijk of de reeks binnen het bereik van een van de vorige hints past en niet groter is dan deze hint. Als dit zo is, pas het bereik aan zodat dit twee vakjes na de reeks eindigt.



Figuur 23. Een rij voor en na het toepassen van regel 2.3, bereik per hint aangegeven.

Regel 2.3 is in het project geïmplementeerd als: *Blackruns IIthree(Blackruns R, char[]line)*

REGELS DEEL 3

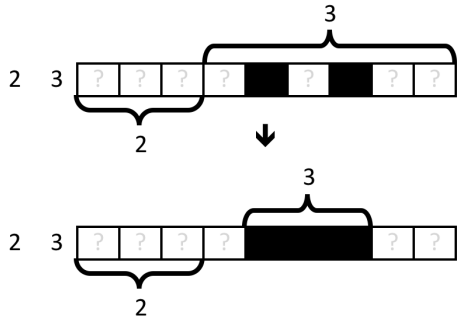
In deel 3 worden zowel vakjes gevuld als het bereik van hints geüpdatet. Deel 3 bestaat uit 3 regels.

Ieder van deze drie regels heeft de volgende vorm:

Blackruns Regel*(Blackruns R, char[]line)*

Er worden twee argumenten meegegeven, *<Blackruns>* en line. line is een *char[]* die de huidige rij/kolom voorstelt. De output van deze functie is diezelfde *<Blackruns>*, maar dan met de regel er op toegepast. Omdat sommige van deze regels ook line aanpassen, kan deze worden opgeslagen in *<Blackruns>*. Na het uitvoeren van de functie, kan de line die een rij uit de puzzel voorstelt, weer in de puzzel worden geplaatst.

Regel 1:
Stel dat in het bereik van een hint meerdere reeksen voorkomen. Indien deze geen deel uit maken van het bereik van een andere hint, vul dan alle onbekende vakjes tussen het eerste en het laatste ingevulde vakje in. Nu kunnen we het bereik updaten. Het begin van het bereik wordt het laatste ingevulde vlakje – (hint – 1). Het einde van de van het bereik wordt het eerste ingevulde vlakje + (hint – 1).



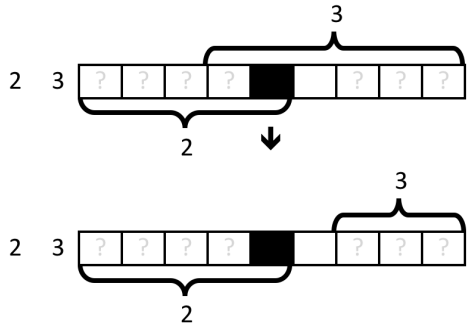
Figuur 24. Een rij voor en na het toepassen van regel 3.1, bereik per hint aangegeven

Regel 3.1 is in het project geïmplementeerd als: *Blackruns IIIone(Blackruns R, char[]line)*

Regel 2:
Stel dat in het bereik van een hint lege vakjes voorkomen. Kijk dan of er genoeg vakjes zitten tussen het begin van het bereik en het eerste lege vakje om een reeks te maken zo lang als de hint. Als dit zo is, stop. Als dit niet zo is, pas het begin van het bereik aan naar het eerst volgende onbekende of ingevulde vakje. Start nu vanaf het begin van het nieuwe bereik en doe hetzelfde.

Hetzelfde maar dan de andere kant op geldt ook, Er moet nu gekeken worden van rechts naar links: Kijk of er genoeg vakjes zitten tussen het einde van het bereik en het eerste lege vakje (vanaf rechts dus) om een reeks te maken zo lang als de hint. Als dit zo is, stop. Als dit niet zo is, pas het einde van het bereik aan naar het eerste onbekende of ingevulde vakje. Start nu vanaf het einde van het nieuwe bereik en doe hetzelfde.

Als er nu reeksen overblijven die kleiner zijn dan iedere hint die deze reeks in zijn bereik heeft, maak dan de cellen van deze reeks leeg.



Figuur 25. Een rij voor en na het toepassen van regel 3.2, bereik per hint aangegeven

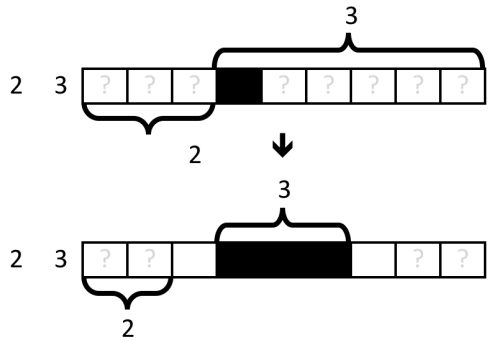
Noot: Het laatste deel van deze regel zouden we weer kunnen zien als een losse regel.

Regel 3.2 is in het project geïmplementeerd als: *Blackruns IIItwo(Blackruns R, char[]line)*

Regel 3:
Regel 3 bestaat op zichzelf weer uit drie deelregels, die er allemaal vanuit gaan dat het bereik van een hint niet overlapt met het bereik van de vorige hint.

Regel 3.1:
Als de het eerste vakje van het bereik van een hint ingevuld is:

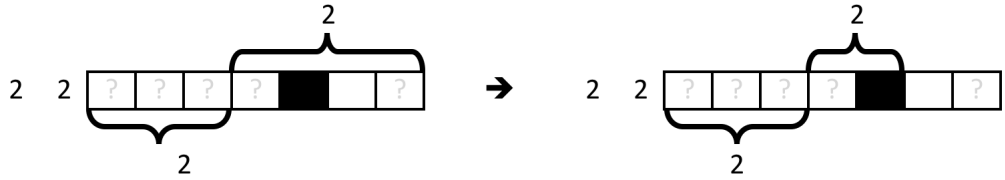
- Vul net zolang cellen in naar rechts totdat we een reeks hebben zolang als de hint;
- Laat voor en na deze reeks een vakje leeg;
- Pas het einde van het bereik aan naar het laatste vakje van de reeks;
- Als het bereik van de volgende hint deze reeks overlapt, pas het beginpunt van dit bereik aan naar het einde van de reeks +2; (dit houdt rekening met het lege vakje tussen iedere reeks)
- Als het bereik van de vorige hint eindigt op het vakje direct voor de reeks, pas dit bereik aan naar twee vakjes voor de reeks (dit houdt rekening met het lege vakje tussen iedere reeks).



Figuur 26. Een rij voor en na het toepassen van regel 3.3.1, bereik per hint aangegeven

Regel 3.3.1 is in het project geïmplementeerd als: *Blackruns IIIthree_one(Blackruns R, char[]line)*

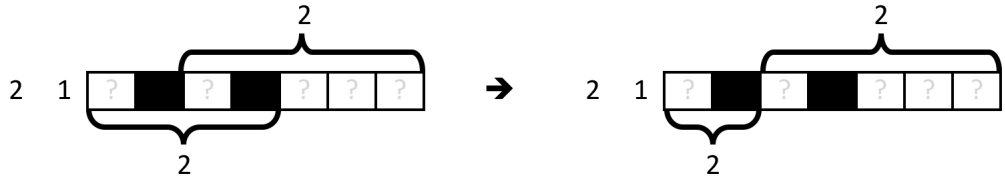
Regel 3.2:
Stel dat binnen het bereik van een hint, een leeg vakje voorkomt na een ingevulde cel. Pas dan het einde van het bereik aan naar het vakje voor dit lege vakje.



Figuur 27. Een rij voor en na het toepassen van regel 3.3.2, bereik per hint aangegeven

Regel 3.3.2 is in het project geïmplementeerd als: *Blackruns IIIthree_two(Blackruns R, char[]line)*

Regel 3.3:
Stel dat binnen het bereik van een hint meerdere reeksen ingevulde vakjes voorkomen. Kijk dan of door het combineren van de eerste reeks met de tweede een reeks zou ontstaan die langer is dan de hint. Als dit zo is, pas het bereik aan naar twee vakjes voor de tweede reeks, stop. Als dit niet zo is, kijk of er nog een volgende reeks is binnen het bereik en pas dezelfde strategie toe door deze te combineren met de eerdere reeksen. Ga zo verder tot de reeks te groot is geworden of het einde van het bereik is bereikt.



Figuur 28. Een rij voor en na het toepassen van regel 3.3.3, bereik per hint aangegeven

Regel 3.3.3 is in het project geïmplementeerd als: *Blackruns IIIthree_three(Blackruns R, char[]line)*

CHRONOLOGISCH BACKTRACKEN

Chronologisch Backtracken (CB) is een vorm van DFS. Aangezien er al een DFS algoritme voor nonogrammen is geïmplementeerd, kan een deel hiervan hergebruikt worden. Bij Chronologisch Backtracken wordt bijgehouden tot welk punt de zoekactie slaagt. Dit heeft als voordeel dat er voortijdig kan worden gestopt met het doorzoeken van een tak die niet tot een oplossing zal leiden.

CB wordt aangeroepen op het moment dat met behulp van de logische regels geen nieuwe informatie meer wordt verkregen. Het doel van CB is het doen van een redelijke gok voor de invulling van een rij. Vervolgens wordt de huidige rij in de puzzel door deze vervangen. Nu worden de logische regels opnieuw uitgevoerd op de puzzel. Als we geluk hebben is de puzzel nu opgelost. Het kan ook zijn dat dit tot een contradictie leidt. Dat wil in het geval van een nonogram zeggen dat een vakje wat leeg is, ingevuld dient te worden of andersom. Als dit het geval is, heeft het invullen van deze oplossing in de rij de puzzel ongeldig gemaakt. Als dit niet tot een contradictie leidt, hebben we een mogelijke oplossing en zullen de volgende rij waar nog onbekende vakjes zijn permuteren en invullen in de puzzel. We gaan zo door tot we alle oplossingen hebben gevonden. Vinden we op deze manier geen oplossing, dan is er simpelweg geen oplossing.

[1] zegt niet hoe om te gaan met een al deels ingevulde puzzel. Dit terwijl CB juist toegepast wordt op puzzels waar de logische regels al op zijn uitgevoerd. In eerste instantie is er vanuit gegaan dat alle permutaties voor een rij worden gegenereerd zonder te kijken naar de huidige invulling. Dit heeft tot gevolg dat grotere puzzels grote hoeveelheden permutaties genereren. Hierdoor duurt het oplossen zeer lang. Vervolgens is besloten om over te gaan op het overslaan van de al ingevulde vakjes. Dit zorgt ervoor dat in veel gevallen de oplostijd drastisch verkort wordt. Hiervoor is permutatepart gemaakt, een aangepaste versie van permute uit DFS.

```
while puzzle not solved
  loop if current line solved
      go to next line
  loop permutations current line
     run logical rules
     if not solved and not inconsistent
        run recursively on next row
```

Figuur 29. Pseudocode van CB.

CHRONOLOGISCH BACKTRACKEN (KLASSE CB)

Set<String> permuatatepart(rulebased info,int[] hints, int length,char[][] puzzle,int line)
De functie permutatepart heeft als invoer vijf argumenten. Het eerste argument is redelijk onbelangrijk, maar zorgt ervoor dat alle informatie beschikbaar is. Heel rulebased wordt meegegeven. De vier andere argumenten zijn delen van rulebased, maar zorgen ervoor dat het geheel overzichtelijk blijft. Het gaat hier over de elementen van een rij: de hints, de lengte van de rij, de puzzel waar de rij in hoort en het rijnummer.

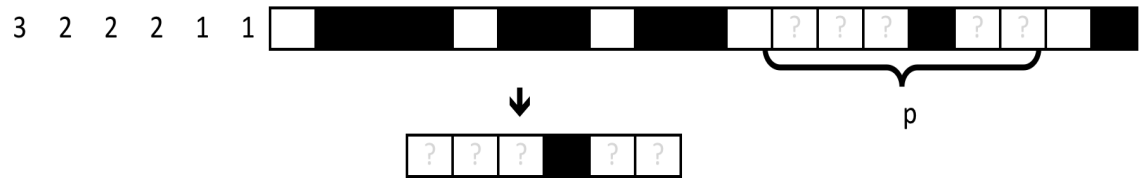
De functie permutatepart zoekt voor de opgegeven line de permutaties. Hierbij wordt alleen het deel dat onbekende vakjes bevat, gepermuteerd. Van de rest is namelijk al bekend hoe de invulling eruit ziet. De functie permutatepart geeft een set van mogelijke rij-invullingen terug.

Set<String> checkpuzzle(char[] line, Set<String> permutations)
De functie checkpuzzle heeft als invoer twee argumenten. “line” is een (deels gevulde) rij uit de puzzel, permutations bevat alle mogelijke invullingen voor diezelfde rij. De functie checkpuzzle vergelijkt de gevonden permutaties met de huidige invulling van de rij uit de puzzel. Incompatibele permutaties worden uit de set gegooid, zodat alleen valide permutaties overblijven.

public boolean solve(Puzzle x)
Deze functie heeft als invoer een *<Puzzle>*. Als uitvoer geeft deze functie false als de puzzel niet oplosbaar is, true als hij opgelost is. In solve wordt volgens chronologisch backtracken geprobeerd de puzzel op te lossen. Dit wordt gedaan door voor de eerste niet volledig ingevulde rij alle mogelijke permutaties te genereren. Vervolgens worden de logische regels losgelaten op de gewijzigde puzzel. Voor elke permutatie die niet leidt tot een oplossing of contradictie wordt solve recursief aangeroepen. Op deze manier worden alle mogelijke oplossingen bezocht en worden alle oplossingen gevonden.

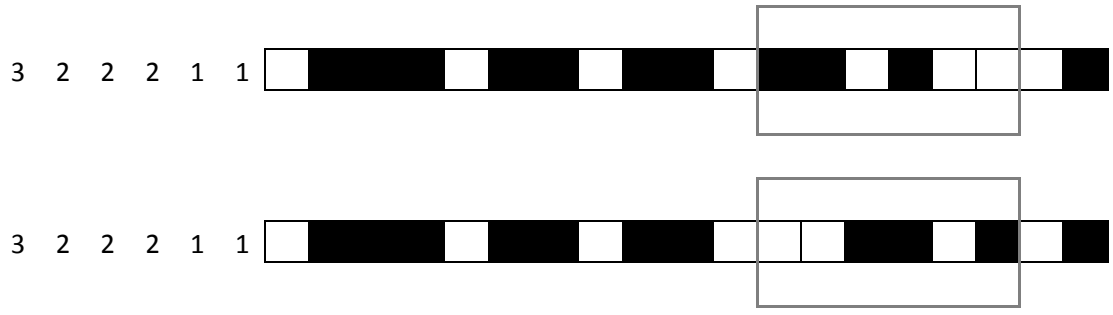
BESCHRIJVING

CB krijgt als invoer een (al deels ingevulde) puzzel mee. Eerst wordt vastgesteld vanaf welke rij er onbekende vakjes voorkomen. Van deze rij wordt het deel tot het eerste onbekende vakje afgeknipt. Voor het deel vanaf het laatste onbekende vakje tot het einde van de rij gebeurt hetzelfde. Dit zorgt ervoor zodat het deel overblijft waar onbekende vakjes in voorkomen.



Figuur 30. Een deels ingevulde rij uit de puzzel en zijn te permuteren deel (p).

Voor dit deel worden alle mogelijke permutaties gecombineerd met het al bekende stuk. Vervolgens wordt door checkpuzzle gekeken welke permutaties matchen met de huidige invulling van de puzzel. Permutaties die niet passen op de huidige invulling worden weggegooid. Dit geldt als een leeg vakje in de permutatie op de plek komt van een al ingevuld vakje in de puzzel. Of als een ingevuld vakje in de permutatie op de plek komt van een leeg vakje in de puzzel. Wat over blijft zijn de permutaties die voldoen aan de hints van de rij en matchen met de huidige invulling.



Figuur 32. De twee permutaties voldoen aan de hints en matchen met de huidige invulling.

De permutaties die nu over blijven dienen als invoer voor chronologisch backtracken. De eerste permutatie vervangt de huidige rij uit de puzzel, waardoor in ieder geval één ingevulde rij ontstaat. De vraag is of dit de juiste invulling voor de rij is. Hier komen we vanzelf achter, door het uitvoeren van de logische regels op de ontstane invulling van de puzzel. Als er tijdens het uitvoeren van één van de regels een contradictie ontstaat, is door het invullen van deze rij de puzzel onoplosbaar geworden.

Als er geen contradictie ontstaat en de puzzel is niet volledig opgelost, beginnen we CB van voor af aan. We zoeken weer de eerste rij die niet volledig is opgelost en genereren daarvoor alle mogelijke oplossingen. Als alle permutaties nu bij het uitvoeren van de regels een contradictie opleveren, dient teruggegaan te worden tot de rij waarop geen contradictie plaatsvond. Hier dient de volgende permutatie te worden geselecteerd. Dit noemen we chronologisch backtracken. Als een oplossing wordt gevonden dient op dezelfde manier te worden verder gezocht als bij een contradictie. In dit geval kan ook niet meer worden verder gezocht, aangezien de puzzel compleet is ingevuld.

RESULTATEN

Een aantal voorbeeldpuzzels is getest met zowel DFS als logische regels. De tabel hieronder geeft de resultaten van één test. De test is uitgevoerd op een iMac medio 2010, 3,06 GHz Intel Core i3 voorzien van 12 GB 1333 MHz DDR3 geheugen en draaiende op Mac OS X Lion 10.7.5 (11G63). De test is uitgevoerd binnen de programmeeromgeving Eclipse Juno. Van twee puzzels is een meervoudige test uitgevoerd. De resultaten hiervan staan in tabel 3.

Puzzel	Aantal oplossingen	Tijd DFS	Tijd Logische regels	CB nodig?
1	1	23 ms	4 ms	Nee
2	1	> min	16 s	Nee
3	1	85 ms	1 ms	Nee
4	1	> min	124 ms	Nee
5	1	> min	2 ms	Nee
6	1	> min	1 ms	Nee
7	2	4 s	11 ms	Ja
8	1	> min	402 ms	Ja
9	1	1 ms	0 ms	Nee
10	8	> min	552 ms	Ja
11	1	> min	1654 ms	Ja
12	1	> min	1447 ms	Nee
13	120	80 ms	443 ms	Ja
14	0	> min	14 ms	Nee
15	1	> min	1249 ms	Ja
16	1	> min	13 s	Nee
17	1	> min	9 s	Nee
18	?	> min	Onbekend > min	Ja

Tabel 2. Resultaten van de single test.

Puzzel	#oplossingen	Gem DFS	STdev	Gem LR	STdev	CB?
3	1	152 ms	21 ms	0,7 ms	0,5 ms	Nee
13	120	85 ms	6 ms	165 ms	19 ms	Ja

Tabel 3. Resultaten van het 10x oplossen van puzzel 3 en 13.

Wat opvalt is dat het oplossen van nonogrammen met behulp van DFS over het algemeen veel meer tijd kost dan met logische regels. De enige uitzondering is puzzel 13, een puzzel met een groot aantal oplossingen. Van deze puzzel is geen enkel vakje op te lossen door logische regels toe te passen. Een groot deel van de permutaties is dan ook direct een geldige oplossing. Het is ook duidelijk dat logische regels niet in staat zijn om een puzzel op te lossen die meerdere valide invullingen heeft. Het feit dat er geen valide puzzel wordt gevonden bevestigd wel dat het algoritme goed werkt. Puzzel 18 is zelfs door het gebruik van logische regels niet snel op te lossen. Dit heeft waarschijnlijk te maken met het aantal oplossingen (minimaal 5) [3] en de grootte van de puzzel.

CONCLUSIE

Er is geprobeerd om aan de hand van [1] een DFS algoritme te maken wat alle mogelijke oplossingen vindt. Ook is geprobeerd om aan de hand van [1] een algoritme te maken wat gebruik maakt van logische regels en vervolgens van chronologisch backtracken om nonogrammen op te lossen. Het is echter op basis van het artikel niet mogelijk om het onderzoek gelijkwaardig over te doen. Daarvoor worden een aantal dingen niet exact genoeg beschreven. Het belangrijkste wat mist, wat van belang is voor zowel DFS als voor CB, is het genereren van mogelijke oplossingen. Er wordt hiervoor geen strategie voorgesteld. Hiervoor is permute bedacht, een functie die vanuit de meest linker oplossing de mogelijke oplossingen van een rij genereert. In het stuk over CB wordt niet gesproken over of en hoe de al deels door logische regels ingevulde puzzel gebruikt wordt voor het genereren van de mogelijke oplossingen voor in de zoekboom. Hiervoor is bedacht om permutaties te genereren en deze te matchen met de huidige invulling van de puzzel.

Een belangrijke stap die [1] niet vermeldt, is dat het mogelijk is een puzzel direct te testen op compatibiliteit door het totaal van de verticale hints te vergelijken met de horizontale hints. Als deze niet overeenkomen hoeft niet eens geprobeerd te worden om de puzzel op te lossen. De puzzel heeft geen oplossing, gezien er voor ieder ingevuld vakje een representatie is in de horizontale en de verticale hints.

Op basis van [1] is het uiteindelijk gelukt om twee werkende oplossers te programmeren. Wat we kunnen concluderen is dat logische regels een goede manier zijn om de zoekruimte van een nonogram te verkleinen. Het is soms zelfs mogelijk om de oplossing van sommige puzzels te vinden door alleen gebruik te maken van logische regels. Helaas gaat dit niet voor alle puzzels op. Dit heeft echter niets te maken met de grootte van de puzzel, maar met de complexiteit ervan.

SUGGESTIES VOOR VERVOLGONDERZOEK

De vraag of nonogrammen met precies één oplossing ook NP volledig zijn, is een interessante. Vervolgens zou gekeken kunnen worden of het mogelijk is de logische regels zo uit te breiden dat alle puzzels met precies één oplossing met alleen logische regels oplosbaar zijn.

In CB valt veel snelheidswinst te behalen. Een idee voor het versnellen van CB is het toevoegen van een gewicht. Dit gewicht wordt gebruikt om te bepalen in welke volgorde de zoekfunctie door de permutaties van de puzzel heen loopt. Een voorbeeld hiervan is het verder zoeken in het pad van de permutatie die het meeste vakjes heeft doen oplossen. Iets anders waar naar gekeken zou kunnen worden, is of de volgorde van de rijen die bezocht worden door CB effect heeft op de snelheid van het oplossen van de puzzel.

Een ander interessant punt is het volgende. Tijdens het oplossen van de puzzel met logische regels wordt voor elke regel door alle rijen van de puzzel heen gelopen. Er is waarschijnlijk snelheidswinst te halen door de regels anders te structureren. Er zou bijvoorbeeld gekeken kunnen worden naar hoe vaak en op welk moment een regel effect heeft op een puzzel. Regel 1.2 heeft bijvoorbeeld geen effect tijdens de eerst iteratie.

REFERENTIES

1. *Yup, Chiung-Hsueh, Hui-Lung Lee, and Ling-Hwei Chen. "An efficient algorithm for solving nonograms." Applied Intelligence 35.1 (2011): 18-31. http://dx.doi.org/10.1007/s10489-009-0200-0*

2. *Jan N. van Rijn, LIACS, 2012 "Playing Games The complexity of Klondike, Mahjong Nonograms and Animal Chess." http://www.liacs.nl/assets/2012-01JanvanRijn.pdf*

3. *Steven Simpson, Lancaster University, Internet site: Nonogram Solver http://www.comp.lancs.ac.uk/~ss/nonogram/ Geraadpleegd 18 juni 2013.*

4. *Tsang EPK (1993) Foundations of constraint satisfaction. Academic Press London*