Computationele Intelligentie
Practicum 1: Backtracking

Wouter Bink, 4105524
Ian van Uunen, 4166132

Deel 1: Backtracking

Een sudoku puzzel wordt gerepresenteerd door een tweedimensionale array van integers, ter grootte van de puzzel. Ook wordt er een Stack gebruikt om successor states in op te slaan.

De volgende methodes worden gedefinieerd als onderdeel van het programma:

- Backtracking(Stack S): Recursively search with a Backtracking Algorithm and return a goal state(completed sudoku).
- IsGoal(int[,] state): Boolean, checks if a given state is a valid goal state.
- Successors(int[,] state): Returns all valid successors to a given state.
- ExpandNext(int[,] state): Chooses the next valid node for expansion.
- sortSuccessors(int[,] state): Sorts all valids successors by domain size. Used in expansion.
- CreateSudoku(string filename): Loads a sudoku from a text file.

```
procedure BackTracking(Stack S):
        If (S.Empty) then return null;
        t = S.Pop();
        If (t.isGoal() or time > timeout) then return t;
        Successors = GetSuccessors(t);
        for successor in successors:
                S.Push(successor)
                T = BackTrack(S);
                If (t.isGoal()) then return t;
endprocedure
```

```
procedure GetSuccessors(t):
        (x, y) = Expand(t);
        illegalValues = new HashSet<int>();
        for all spaces in row y
                If the space is not empty
                        then illegalValues ← value of the space
        for all spaces in column x
                If the space is not empty
                        then illegalValues ← value of the space
        for all spaces in the (√N x √N) region
                If the space is not empty
                        then illegalValues ← value of the space
        Successors = array of Type t
        for all values from 1 to N
                if illegalValues does not contain value then
                        newState = t.Clone();
                        newState(x,y) = value;
                        Successors ← newState;
        Return Successors;
endprocedure
```

```
procedure isGoal(t):
        for all spaces in t
                if space is not empty then continue;
                else return False;
        return True;
endprocedure
```

```
procedure Expand(t)
switch(expandmethod)
        case LeftToRight:
        (i,j) = (1,1);
        while t(i, j) is not empty and i < N
                if j < N  then j++;
            else j = 1; i++;
        endwhile;
       return (i, j);
```

```
        case RightToLeft:
        (i, j) = (N, N);
      while  t(i,j) is not empty and i >= 1
        if j > 1 then j--;
            else j = N; i--;
        endwhile;
      return (i, j);

        case domainSize:
        return  ExpansionPriority.First(); *
endprocedure
```

**with ExpansionPriority being a descending priority list of spaces sorted on the amount of illegal values for that space, computed from the initial sudoku.*

Table 1: Backtracking runtime and recursion count for different sudokus.

Puzzel		**Left to right**	**Right to left**	**Domain Size**
Standaard puzzel	Runtime: Recursion:	168,370 ms 69,175,317	1,220 ms 518,275	29,156 ms 13,705,736
Project Euler Grid 01	Runtime: Recursion:	5 ms 201	5 ms 541	22 ms 275
Project Euler Grid 02	Runtime: Recursion:	8 ms 295	19 ms 4277	28 ms 2,838
Project Euler Grid 29	Runtime: Recursion:	49 ms 14,955	24 ms 5,246	99 ms 20,923
Project Euler Grid 36	Runtime: Recursion:	5 ms 238	4 ms 84	31 ms 81
Project Euler Grid 50	Runtime: Recursion:	46 ms 16,162	16 ms 4,585	50 ms 8,447
Sfsudoku Line 01	Runtime: Recursion:	494 ms 204,901	2,984 ms 1,391,110	1,701 ms 706,295
Sfsudoku Line 02	Runtime: Recursion:	374 ms 130,495	118,159 ms 56,116,919	7,611 ms 3,113,570
Sfsudoku Line 03	Runtime: Recursion:	13,047 ms 5,499,660	12,436 ms 5,867,837	6,227 ms 2,768,022

Sfsudoku Line 499	Runtime: Recursion:	10,187 ms 4,166,646	7,262 ms 3,182,617	146 ms 46,182
Sfsudoku Line 500	Runtime: Recursion:	643 ms 235,915	10,253 ms 4,757,416	42,573 ms 19,556,897
16x16_1 vakpagina	Runtime: Recursion:	57,328 ms 14,579,935	146 ms 42,435	6,282 ms 1,621,134
16x16_2 vakpagina	Runtime: Recursion:	Time limit	Time limit	Time limit
16x16_3 vakpagina	Runtime: Recursion:	Time limit	Time limit	Time limit

Onze backtracking was niet snel genoeg voor de moeilijkere 16x16 puzzels
(De snelste tijd en minste recursieve aanroepen zijn groen gekleurd in de tabel)

Conclusies

De methode met de minste recursieve aanroepen is vrijwel altijd het snelst. Hieruit blijkt dat de tijdsduur van het algoritme sterk afhankelijk is van het aantal aanroepen. De enige uitzondering hierop is de euler puzzel 36, waarbij de domain size het minste aanroepen heeft, maar wel het langste duurt. Dit komt door de korte tijd waarin deze eenvoudigere puzzel opgelost kan worden. Waarbij Domain Size wel eerst nog wat tijd nodig heeft of de hokjes die hij gaat expanden te sorteren. Ook is te zien dat hij bij alle eenvoudige euler puzzels het langzaamst is, ook al heeft hij minder recursieve aanroepen dan een ander.

Wel is te zien dat de Domain Size methode consistenter is in het vinden van een kortere oplossing. Bij de andere twee methodes is het aantal recursieve aanroepen sterk afhankelijk van de layout van de puzzel en moet men geluk hebben er voordelig uitziet voor de methode.
Als we het totaal van aanroepen bij elkaar optellen komt:
LeftToRight op ongeveer 94 miljoen;
RightToLeft op ongeveer 72 miljoen;
DomainSize op ongeveer 41 miljoen.
Met wat geluk zijn LTR en RTL voor een aantal puzzels sneller dan DS, maar gemiddeld is DS toch ongeveer twee keer zo snel als LTR en RTL, op deze kleine sample set van puzzels.

Deel 2: Forward Checking

Voor de datastructuur hebben wij meerdere verschillende systemen getest. Hierbij kwam een 2d int[,] array van values en een 2d List<int>[,] array van domeinen, als snelste uit de test. Deze versloeg onder andere de Field[,] array (waar Field een class is die value en domein bevat) en de bool[,][] array waar een waarde van *true* betekend dat dat getal in het domein staat.
Buiten de functies in Deel 1, hebben wij de nieuwe functie *Forward Checking* die kijkt of er geen domein leeg wordt als we de waarde newValue op positie (x, y) proberen in te vullen:

---

procedure ForwardChecking(domains, (x, y), newValue):
    for *all spaces in row y*
        If *the domain of the space counts only one* **and** *the value of the domain == newValue* **then** return false;
    for *all spaces in column x*
        If *the domain of the space counts only one* **and** *the value of the domain == newValue* **then** return false;
    for *all spaces in the (√N x √N) region*
        If *the domain of the space counts only one* **and** *the value of the domain == newValue* **then** return false;
    return true;
endprocedure

Daarnaast hebben we de nieuwe functie *RemoveFromDomain,* die als *Forward Checking* true returned, de newValue uit het domein van de juiste hokjes weghaalt:

---

procedure RemoveFromDomain(domains, (x, y), newValue):
    for *all spaces in row y*
        remove newValue from the space's domain
    for *all spaces in column x*
        remove newValue from the space's domain
    for *all spaces in the (√N x √N) region*
        remove newValue from the space's domain
endprocedure

En als laatste is er een kleine aanpassing voor de *Expand* functie, zodat deze ons, net als bij de DomainSize case, het eerste vakje van een priority list geeft, die gesorteerd is het op aantal illegale successors. Maar nu wordt het aantal successors, de domein grootte, hiervoor berekend aan de hand van

de huidige state, in plaats van alleen de initiële sudoku state.

Table 2: Forward checking runtime and recursion count for different sudokus.

Puzzel		**Forward Checking**	**Fastest BT**
Standaard puzzel	Runtime: Recursion:	519 ms 39,223	1,220 ms 518,275
Project Euler Grid 01	Runtime: Recursion:	8 ms 50	5 ms 201
Project Euler Grid 02	Runtime: Recursion:	10 ms 56	8 ms 295
Project Euler Grid 29	Runtime: Recursion:	14 ms 376	24 ms 5,246
Project Euler Grid 36	Runtime: Recursion:	8 ms 50	4 ms / 31 84/ 81
Project Euler Grid 50	Runtime: Recursion:	10 ms 156	16 ms 4,585
Sfsudoku Line 01	Runtime: Recursion:	30 ms 1,589	494 ms 204,901
Sfsudoku Line 02	Runtime: Recursion:	99 ms 5,203	374 ms 130,495
Sfsudoku Line 03	Runtime: Recursion:	491 ms 38,927	6,227 ms 2,768,022
Sfsudoku Line 499	Runtime: Recursion:	70 ms 5,109	146 ms 46,182
Sfsudoku Line 500	Runtime: Recursion:	88 ms 6,393	643 ms 235,915
16x16_1 vakpagina	Runtime: Recursion:	30 ms 346	146 ms 42,435
16x16_2 vakpagina	Runtime: Recursion:	Time limit	Time limit
16x16_3 vakpagina	Runtime: Recursion:	Time limit	Time limit

Conclusies

Backtracking met Forward-Checking laat over het algemeen een significant kortere tijd zien dan de snelste van de drie in Deel 1. Ook is het aantal recursies veel kleiner, aangezien een groot aantal mogelijke vertakkingen al vroeg gesnoeid worden als FC een domein leeg maakt. Ondanks dat is onze implementatie nog steeds onvoldoende om de moeilijkere 16x16 sudokus binnen de tijdslimiet op te lossen. In een poging om dit te verhelpen hebben wij de volgende optimalisaties doorgevoerd:

- Verschillende datastructuren (zoals hierboven uiteengezet)
- Uitstellen van expansie d.m.v. een enumerable/yield Iterator Pattern.
- Minimaliseren van DeepClone() aanroepingen door eerst een gedeeltelijke FC uit te voeren.

Hoewel wij hier nog een significante (factor 2-3) verbetering door zagen, was dit nog steeds niet genoeg. Uit Profiling bleek dat het overgrote deel van de tijd besteed werd in de bovengenoemde Clone methode, welke wij niet verder wisten te versnellen in de tijd die wij hadden.