Implementatie chronologische BackTracking algoritme

Ons chronologische backtracking algoritme werkt met deze stappen:

1. Verkrijg het bord dat opgelost moet worden. (Solve Methode)
2. Maak het bord consistent. (Solve Methode)
3. Verkrijg een lijst met alle vlakken die leeg zijn en ingevuld dienen te worden. Deze lijst wordt zodanig geordend dat de volgorde van vlakken in de lijst van links naar rechts, rechts naar links, of geordend op domein grootte zijn. (Solve Methode)
4. Voor backtracking uit het op bord, met als start positie het eerst vlak uit de lijst te doorzoeken vlakken. (Backtrack Methode)
5. Backtracking geeft True of False terug. True als het bord is opgelost, false als dat niet is gelukt. Als true wordt teruggegeven door backtrack dan wordt het opgeloste bord teruggegeven, zo niet wordt null teruggegeven. (Solve Methode)

Hieronder in detail wordt met pseudo-code besproken:

- Hoe het bord consistent wordt gemaakt.
- Hoe de lijst van te doorzoeken vlakken verkregen wordt.
- Hoe backtracking wordt gedaan en de daartoe behoren hulp methodes.

Solve Methode:

```
// Geef aan deze methode, een bord dat nog niet is opgelost, in welke volgorde vlakken
doorzocht moeten worden als string
Solve(Board board, string searchMode) return board
{
	MakeStartingBoardConsitent(board);

	SquaresToSolve = FindSquaresToSolve(board, searchMode);

	startingSquare = SquaresToSolve[first element]

	if (backtrack(startingSquare))
		return board;

	return null;
}
```

Hulp Methodes voor Solve

- MakeStartingBoardConsistent
- SquaresToSolve

MakeStartingBoardConsistent Methode:

```
// Er wordt naar elk vlak gekeken, en als en dit vlak maar één getal staat, wordt dit
getal weg gehaald uit alle buren/peers van het vlak.
MakeStartingBoardConsitent(Board board)
{
	For each (square in board)
	{
		if (possible values in square == 1)
		{
			// vlak heeft meer dan een mogelijkheid, dus pak het eerste
			element uit de lijst van mogelijke getallen van het vlak.
			numberToRemove = square[0];

			for each (peer in square's peers)
			{
				peer's Values.Remove(numberToRemove);
			}
		}
	 }
}


// Maak een lijst aan van alle nog niet opgeloste vlakken en zit die in een bepaalde
volgorde.
SquaresToSolve(Board board, string searchMode)
{
	squareList = List<string>;

	// elk vlak waarin meer dan één getal staat is niet opgelost en moet in de te
	doorzoeken lijst komen
	foreach (square in board)
	{
		if (square.Value.Count > 1)
			squareList.Add(square);
	}

	// als er van recht naar links gezocht moet worden, draai de lijst om.
	if (searchMode == right to left)
		squareList.Reverse();
	// als er geozocht moet worden op basis van domeingrootte.
	elif (searchMode == "domain size")
		orderByDomainSize(board);
}
```

```
Bool backtrack (string position)
{
	//If the position == null, this means all the unfixed squares have been filled in
	This automatically means the board is solved, because it can't get to this line of code before
	running through the allowed() function
	If (position == null)
		Return true;

	//Check for each number if it is possible to place it in the position you are currently working
	with, where allowed takes the position and the board to see if its peers allow for this
	List Number = getPossibleNumbers
```

```
        For each (number x)
        {
                Board[position] = x;
                If ( allowed(board, position))
                        If (backtrack (nextPosition))
                                Return true;
        }

        //If there is not a single number allowed in this location, you have reached the end of a
        branch and need to backtrack until you find an unsearched path
        Board[position] = null;
        Return false;
}
```