HTL OTTAKRING

Das Damenproblem

Lösen des Problems in C#

Lorenz Leutgeb und Moritz Wanzenböck

Arbeitsbeginn am 01.10.2009

Projektabgabe am 12.11.2009

Abgabe an Ing. Dipl.-Päd. Michael Rausch

Inhaltsverzeichnis

1 Angabe

Es gilt n Damen auf einem Schachbrett der Größe $n * n$ so zu platzieren, dass sie sich gegenseitig nicht bedrohen, also keine Dame eine andere in einem Zug schlagen kann.

Dabei sind die Zugregeln des Schachspiels zu beachten, laut welchen eine Dame horizontal, vertikal und diagonal ziehen, aber nicht über eine andere Figur springen kann.

2 Formulierung eines Algorithmus

2.1 Grundlegende Überlegungen

Um einen Algorithmus, welcher dieses Problem lösen kann zu formulieren, muss eine Methode gefunden werden, welche feststellt ob ein Feld bedroht ist oder nicht. Diese Information muss beim Platzieren der Damen berücksichtigt werden.

2.2 Verfeinerung

Ein Bruteforce Algorithmus, welcher das Schachbrett durchläuft und auf das erstbeste nicht bedrohte Feld eine Dame setzt, kann dieses Problem nicht lösen, da auf diese Art und Weise kein freies Feld mehr übrig ist, bevor die Anzahl der Damen n erreicht. Es muss also eine Methode gefunden werden, welche erkennen kann, dass kein nicht bedrohtes Feld mehr verfügbar ist und in diesem Fall die zuvor platzierte Dame auf ein anderes Feld setzt. Besonders gut eignet sich die Backtracing Methode, welche nach dem „Trial and Error“ Prinzip vorgeht, um eine Lösung zu finden. Im Falle des Damenproblems wird also versucht eine Dame zu platzieren (Trial) und solange fortgefahren bis ein Fehler entdeckt (Error), oder eine Lösung gefunden wird. Einen solchen Algorithmus iterativ zu formulieren ist sehr aufwändig, da ein Zwischenspeicher benötigt wird um die Operationen aufzuzeichnen, die im Falle eines Fehlers rückgängig gemacht werden müssen. Die rekursive Formulierung ist besser geeignet, da eine Aufzeichnung durch die Rekursionsebenen gegeben ist.

2.3 Pseudocode

Eine Formulierung des Algorithmus als Pseudocode:

```
funktion setzedame(zeile, spalte)
{
  wenn(zeile == n + 1)
    ende ok;

  wenn(feldsicher(zeile, spalte))
    setzedame(zeile, spalte)

  schleife(variable i von 0 bis n)
    wenn(setzedame(zeile + 1, i) == ok)
      ende ok;

  entfernedame(zeile, spalte);

  ende fehler;
}
```

Um eine korrekte Terminierung zu gewährleisten muss zu allererst überprüft werden ob bereits n Damen platziert wurden. Wenn das Programm versucht in der Zeile $n+1$ eine Dame zu setzen, ist eine Lösung gefunden, andernfalls muss mit dem Lösen fortgefahren werden.

Falls die Anzahl der Damen noch nicht n erreicht hat muss überprüft werden ob das aktuell in Bearbeitung befindliche Feld bedroht wird. Welches Feld überprüft werden soll wird dabei von der vorigen Rekursion bestimmt. Wenn das Feld nicht bedroht wird, ist eine Dame zu platzieren und eine Schleife zu starten die für jede Spalte der nächsten Zeile eine Rekursion auslöst, da Zeile n schon bedroht ist, aber noch nicht alle Spalten.

Wenn für kein Feld der Zeile $n+1$ eine Lösung gefunden wird muss die Dame in der aktuellen Zeile falsch platziert sein, es wird also die Dame entfernt und eine Rekursionsebene zurückgesprungen.

2.4 Überprüfungsmethode

Damit der Kern des Algorithmus richtig funktionieren kann, muss die Funktion `feldsicher()`, welche unter Punkt 2.3 verwendet wird, definiert werden. Diese Funktion überprüft, ob ein Feld bereits von einer Dame am Spielfeld bedroht wird. Zur Erläuterung:

```
funktion setzedame(zeile, spalte)
{
   schleife(variable i von 0 bis n)
     wenn(dame auf feld(i, spalte))
       ende bedroht;

   schleife(variable i = x-1, j = y-1 solange i > -1 und j > -1)
     wenn(dame auf feld(j, i))
       ende bedroht;

   schleife(variable i = x+1, j = y+1 solange i < n && j < n)
     wenn(dame auf feld(j, i))
       ende bedroht;

   schleife(variable i = x+1, j = y-1 solange i < n und j > -1)
     wenn(dame auf feld(j, i))
       ende bedroht;

   schleife(variable i = x-1, j = y+1 solange i > -1 && j < n)
     wenn(dame auf feld(j, i))
       ende bedroht;

}
```

Die Funktion startet mit der Überprüfung der Spalte auf eine Dame. Sie geht also die entsprechenden Felder des Schachbretts durch und überprüft ob eine Dame am jeweiligen Feld steht. In diesem Fall ist das Feld bedroht und die Funktion wird beendet.

Danach folgt die diagonale Überprüfung. Zuerst werden die Felder „links über“ dem zu überprüfenden Feld abgearbeitet, dann die „rechts darunter“. Dies wird über zwei getrennte Schleifen erledigt.

Wenn noch keine Bedrohung festgestellt wurde, wird zur letzten Überprüfung fortgefahren welche „links unter“ und „rechts über“ dem zu überprüfenden Feld nach Damen sucht.

Wurde noch immer keine Bedrohung gefunden, ist das Feld sicher und die Funktion endet mit entsprechendem Rückgabewert.

3 Das Programm

Das Programm kann Lösungen für das Damenproblem finden und auf einem zweidimensionalen „Schachfeld“ anzeigen. Das Programm ist auf Problemstellungen des Ausmaßes von $n < 28$ limitiert, da die Berechnungen größerer Werte auf aktuellen Rechnern nicht realisierbar sind. Eine Ausgabe als Bild- sowie Textdatei ist integriert.

Gefundene Lösungen können angezeigt werden indem sie in der `ListBox` ausgewählt werden.

3.1 Optionen

Die Optionen können über *Datei – Optionen* aufgerufen werden:

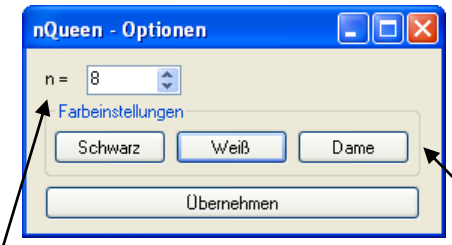

Abb. 1 (Optionen)

Im Dialogfeld kann n angepasst werden. Eine Veränderung der Spielbrettfarben, sowie der Damen ist möglich (zum Beispiel bei Kontrastproblemen).

Sobald der Dialog geschlossen wird, werden die Einstellungen in den Hauptdialog übernommen und das Schachbrett neu gezeichnet.

Siehe Settings.cs und Main.cs Zeilen 164-168

3.2 Exportieren

Das Programm kann eine Lösung als Textdatei oder Grafik exportieren. Die Funktionen können mit den Menüpunkten *Exportieren – Textdatei* bzw. *Exportieren – Grafik* genutzt werden.

3.2.1 Textdatei

Nach dem Aufruf von *Exportieren – Textdatei* sind die Platzhalter für besetzte Schachfelder auszuwählen. Außerdem muss der Speicherort der Textdatei angegeben werden.

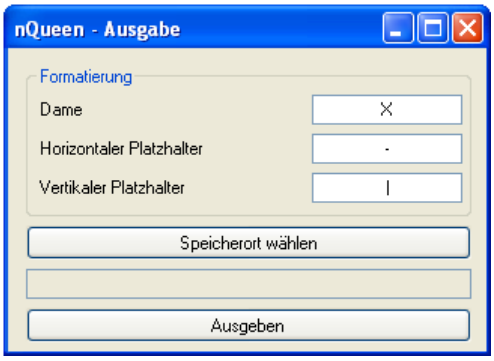

Abb. 2 (Exportieren - Textdatei)

Siehe OutputText.cs und Main.cs Zeilen 170-174

3.2.2 Grafik

Sobald im Menü *Exportieren* der Punkt *Grafik* aufgerufen wird, ist in einem Dialog der Speicherort der Grafik anzugeben.

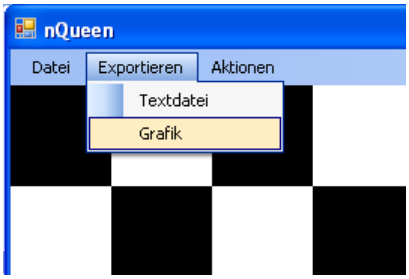

Abb. 3 (Exportieren - Grafik)

Siehe Main.cs Zeilen 176-256

3.3 Aktionen

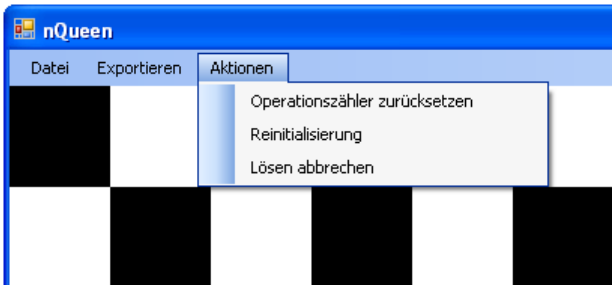

Abb. 4 (Aktionen)

Die Funktionen des Untermenüs *Aktionen* dienen dem Rücksetzen des Operationszählers bzw. dem neu Zeichnen des Schachbrettes und dem Löschen aller aktuell gefundenen Lösungen.

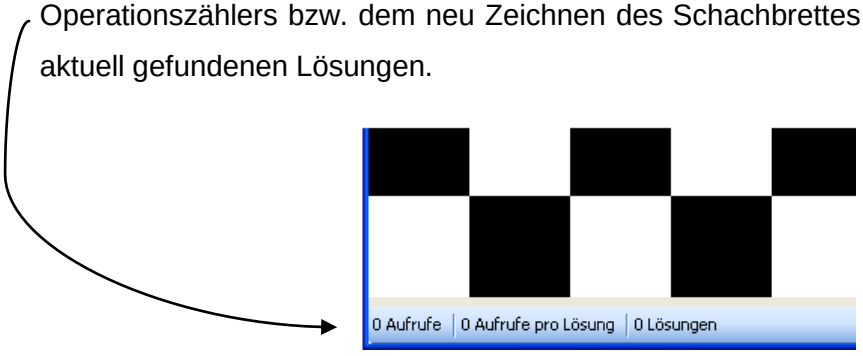

Abb. 5 (Operationszähler)

4 Programmcode

4.1 Starten des Lösevorgangs und Speichern der Lösungen

Die Methode `startSolve()` zum Starten des Lösungsvorgangs wird durch klicken auf *Datei – Lösen* aufgerufen. Sie startet einen neuen Thread und beginnt daraufhin gefundene Lösungen in die `ListBox` aufzunehmen.

```
private void startSolve(object sender, EventArgs e)
{
   WorkThread = new Thread(new ThreadStart(solve));
   WorkThread.Start();

   int SolutionCount = 0;
   while (SolutionCount < BoardQueue.Count || WorkThread.IsAlive)
   {
      Application.DoEvents();

      if (SolutionCount < BoardQueue.Count)
         listBox.Items.Add("Lösung " + ++SolutionCount);

      callsLabel.Text = Convert.ToString(h + " Aufrufe");

      if(listBox.Items.Count > 0)
         callsPerSolutionLabel.Text = Convert.ToString(h/listBox.Items.Count
```

```
            + " Aufrufe pro Lösung");

        solutionsLabel.Text = Convert.ToString(listBox.Items.Count
            + "Lösungen");
    }

    listBox.SelectedIndex = 0;
}
```

Nach Initialisieren und Starten des Threads wird zu einer `while` Schleife fortgefahren welche die eigentliche Schnittstelle zwischen dem rechnenden Thread und dem Formular bildet. Sie prüft vor jedem Durchgang ob es noch Lösungen im Zwischenspeicher gibt, oder der zuvor gestartete Thread noch arbeitet. Innerhalb der Schleife werden nun mit `Application.DoEvents()` alle GUI-Events abgearbeitet. Falls nun die Anzahl der Lösungen im Zwischenspeicher größer ist als die Anzahl der Einträge in der `ListBox`, wird ein neuer Eintrag an die `ListBox` angehängt. Zuletzt werden die Labels zur Statusinformation aktualisiert. Sobald der Thread seine Berechnungen fertiggstellt hat, wird die erste Lösung angezeigt.

4.2 Anstoßen des Lösevorgangs

```
private void solve()
{
    for (int i = 0; i < WorkBoard.Length ; i++)
        set(WorkBoard, 0, i);

    Thread.CurrentThread.Abort();
}
```

Der WorkThread startet standardmäßig die Methode `solve()` welche wiederum für jede Spalte der ersten Zeile des Schachfeldes eine Lösung sucht. Dies wird dadurch erreicht, dass die Funktion `set()`, welche den eigentlichen Algorithmus beinhaltet, immer mit der Zeile 0 und der Spalte i „angestoßen“ wird, solange $i \leq n$ gilt. Wenn Spalte n erreicht und gelöst ist, beendet sich der Thread.

4.3 Lösevorgang

In dieser Methode befindet sich der eigentliche Algorithmus zum Lösen des Damenproblems. Als Parameter dient das zu bearbeitende bool-Array und die Indizes der zu platzierenden Dame. Die zu übergebenden Variablen sind `bool[][]` t, das Schachbrett und die Koordinaten des zu bearbeitenden Feldes. x repräsentiert die Spalte und y die Zeile.

```
private bool set(bool[][] t, int y, int x)
{
```

```
      h++;

      if (!safe(t, y, x))
         return false;

      t[y][x] = true;

      if (y + 1 >= t.Length)
         lock (queue)
         {
            BoardQueue.enqueue(t);
         }

      for (int i = 0; i < t.Length; i++)
         if (set(t, y + 1, i))
            return true;

      t[y][x] = false;

      return false;
   }
```

Zu Beginn jeder Rekursion bzw. dem Start des Algorithmus wird `Operations` erhöht. Dieser Wert dient zur Rückmeldung an den Benutzer des Programms. Es folgt die Überprüfung des aktuellen Feldes (siehe 4.4). Falls das Feld bedroht ist, wird die Rekursion abgebrochen und `false` zurückgegeben, was eine Korrektur der vorher platzierten Dame zur Folge hat. Wenn jedoch keine Bedrohung vorliegt, wird eine Dame platziert. Nun wird überprüft ob eine Lösung gefunden ist. In diesem Fall wird der Zugriff auf den Zwischenspeicher blockiert und die Lösung gespeichert. Das Blockieren des Zwischenspeichers für den Zugriff des Threads ist zwingend notwendig, da sonst die Wahrscheinlichkeit einer Kollision mit dem auslesenden Thread viel zu hoch wäre. Der Algorithmus überprüft nun ob die gerade platzierte Dame richtig steht. Es wird versucht in jeder Spalte der nächsten Zeile eine Dame zu platzieren. Wenn dies bei einem Feld gelingt, steht die Dame richtig und `true` wird zurückgegeben. Liefert keiner der Versuche ein positives Ergebnis steht die Dame falsch. Sie wird wieder entfernt und `false` zurückgegeben.

4.4 Überprüfung

Die Überprüfung findet in der Methode `safe()` statt. Die zu übergebenden Variablen sind `bool[][]` t, das Schachbrett und die Koordinaten des zu überprüfenden Feldes. x repräsentiert die Spalte und y die Zeile.

```
   private bool safe(bool[][] t, int y, int x)
   {

      int i, j;

      for (i = 0; i < t.Length; i++)
         if (t[i][x])
            return false;
```

```
      for (i = x - 1, j = y - 1; i > -1 && j > -1; i--, j--)
         if (t[j][i])
            return false;

      for (i = x + 1, j = y + 1; i < t.Length && j < t[i].Length; i++, j++)
         if (t[j][i])
            return false;

      for (i = x + 1, j = y - 1; i < t.Length && j > -1; i++, j--)
         if (t[j][i])
            return false;

      for (i = x - 1, j = y + 1; i > -1 && j < t[i].Length; i--, j++)
         if (t[j][i])
            return false;

      return true;
   }
```

Nach der Initialisierung der zwei Laufvariablen i und j wird die Überprüfung der Spalte gestartet. Die Spalte bleibt konstant bei x während i für die Zeile eingesetzt wird solange $i \leq n$ gilt. Sobald eine Zelle `true` enthält, liegt eine Bedrohung vor und die Methode endet mit dem Rückgabewert `false`. Eine Überprüfung der Zeile ist nicht notwendig, da hier keine Bedrohung vorliegen kann. Dies wird dadurch garantiert, dass das Schachfeld beim Suchen einer Lösung von oben nach unten abgearbeitet wird und von `set()` niemals auf den Bereich unter der zu bearbeitenden Zeile y zugegriffen wird. Es folgt die Überprüfung der Diagonalen. Zuerst werden alle Felder mit Koordinaten $0 \leq i < x$ und $0 \leq j < y$ überprüft. Dies deckt alle Felder auf der Diagonale „links über“ dem Feld $(y|x)$ ab. Diese Schleife wird im gleichen Schema für $x < i \leq n$ und $y < j \leq n$ erneut durchgeführt . Kombiniert ergibt dies eine Überprüfung aller Felder der Diagonale von $(0|0)$ bis $(n|n)$. Der gleiche Vorgang wird nun gespiegelt wiederholt um die Diagonale von $(n|0)$ nach $(0|n)$ abzudecken. Enthält keines der Felder $(j|i)$ `true`, wird `true` zurückgegeben.

5 Die Klasse MoQueue

`MoQueue` dient im Programm als Zwischenspeicher der Lösungen. Der Zwischenspeicher besteht aus einem zweidimensionalen Array des Datentyps `Queue`:

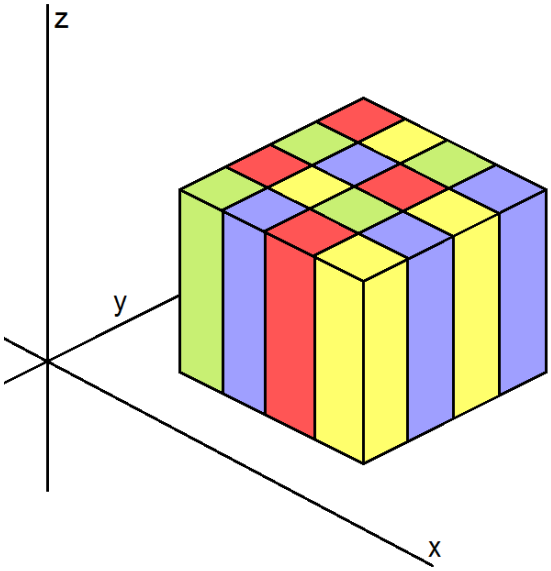

Abb. 6 (MoQueue Skizze)

Die Lösungen werden also wie in der Skizze in Stapeln entlang der z-Achse angeordnet. x und y entsprechen den Koordinaten am Schachbrett. Insgesamt gesehen ergibt sich daraus ein Quader mit den Maßen $n * n * s$, wobei s der Anzahl der gefundenen Lösungen entspricht.

5.1 Der Konstruktor

```
public MoQueue(int n)
{
   this.n = n;
   q = new Queue[n][];

   for (int i = 0; i < n; i++)
   {
      q[i] = new Queue[n];
      for (int j = 0; j < n; j++)
         q[i][j] = new Queue();
   }
}
```

Das zweidimensionale Array q vom Typ `Queue` der Größe $n * n$ wird erstellt und n für die spätere Verwendung durch andere Member abgespeichert.

5.2 Count

`Count` repräsentiert die Länge von q und somit die Anzahl der Lösungen im Zwischenspeicher.

5.3 Enqueue

`Enqueue()` dient dem Abspeichern eines zweidimensionalen Arrays vom Typ `bool`. Das Array, das an die bestehende Schlange angehängt wird ist als Parameter an die Methode zu übergeben.

```
public void Enqueue(bool [][] x)
{
   Count++;

   for (int i = 0; i < x.Length; i++)
      for (int j = 0; j < x[i].Length; j++)
         q[i][j].Enqueue(x[i][j]);
}
```

x, das zu speichernde Array (Lösung) wird durchgearbeitet und Zelle für Zelle in die `Queues` von q gespeichert.

5.4 Dequeue

Die Klassenmember `Dequeue()` vom Typ `bool[][]` liefert das erste abgespeicherte Array zurück.

```
public bool[][] Dequeue()
{
   bool[][] r = new bool [n][];

      for (int i = 0; i < r.Length; i++)
         r[i] = new bool[n];

   Count--;

   for (int i = 0; i < r.Length; i++)
      for (int j = 0; j < r[i].Length; j++)
         r[i][j] = (bool)q[i][j].Dequeue();

   return r;
}
```

Dazu wird zunächst ein leeres zweidimensionales Array mit der Größe $n * n$ initialisiert. Anschließend wird `Count` dekrementiert. Das Array wird nun von zwei Schleifen, welche

jede `Queue` ansprechen und den jeweils ersten Wert abrufen befüllt. Der letzte Schritt besteht in der Rückgabe des gefüllten Arrays.

5.5 GetArray

Die Funktion `GetArray()` vom Typ `bool[][]` dient dem Auslesen eines Arrays an einer beliebigen Stelle der `Queue`. Die Stelle wird über den Parameter `int` x übergeben.

```
public bool[][] GetArray(int x)
{
   bool[][] r = new bool[n][];

   for (int i = 0; i < n; i++)
      r [i] = new bool[n];

   for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++)
         r [i][j] = (bool)q[i][j].ToArray()[x];

   return r;
}
```

Eine Schleife erzeugt ein Array r vom Typ `bool[][]` der Größe $n * n$. Nun wird jede `Queue` in q aufgerufen, in ein Array umgewandelt und der Wert an Position x des Arrays in r eingefügt. Das gefüllte Array wird zurückgegeben und die Funktion endet.

Abbildungsverzeichnis