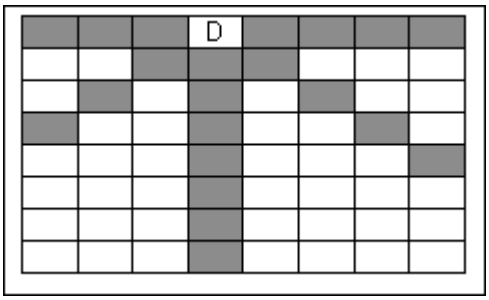
Das Problem der acht Damen



Im folgenden Text soll nun gezeigt werden, dass man auch mit den einfachen Suchverfahren der Tiefensuche oder Breitensuche das gegebene Problem lösen kann, aber es soll auch die prinzipielle Erweiterbarkeit des Problems mit den sich daraus ergebenden prinzipiellen Problemen betrachtet werden.

Datenstruktur

In Python ist der naheliegende Ansatz eine Liste von Koordinatenpaaren, welche die Position der gesetzten Damen angeben. Eine Liste der Form

[[1 , 1] , [2 , 5] , [3 , 8] , [4 , 6] , [5 , 3] , [6 , 7] , [7 , 2] , [8 , 4]]

beschreibt dann eine mögliche Belegung des Feldes mit Damen. Ob diese allerdings zulässig ist, sei hier noch dahin gestellt.

Eine Primitivlösung

Nun ergibt sich nach dem Prinzip des ***Generate and Test*** folgende sehr primitive *„Lösung"* des Problems:

- Erzeuge als Grundmenge eine Liste aller möglichen Koordinatenpaare mit den Werten von 1..8.
- Generiere z.B. mit Hilfe einer Tiefensuche mit backtracking alle möglichen Kombinationen von genau 8 dieser Koordinatenpaare.
- Teste jede generierte 8 – elementige Liste von Koordinatenpaaren, ob die von ihr beschriebene Belegung zulässig ist.
- Führe das durch, bis (*bei einfacher Suche*) eine Belegung zulässig ist oder (*bei vollständiger Suche*), bis alle Möglichkeiten untersucht wurden.

Nach diesem Prinzip arbeitet das folgende Programm:

```
def suche(alternativen, belegt):
    if len(belegt)==maxZahl:
        if zulaessig(belegt, belegt[1:]):
            return belegt
        return False
    if len(alternativen)==0:
       return False
    versuch = suche(feld, [alternativen[0]]+belegt)
    if versuch:
        return versuch
    else:
        return suche(alternativen[1:], belegt)
```

Startet man dieses Programm, arbeitet es bei acht Damen „*sehr lange*". Warum?

Der entscheidende Nachteil dieser Version ist, dass als Alternativen alle Felder des Spielfeldes versucht werden. Tiefensuche ist sowieso schon ein uninformiertes Suchverfahren, aber diese Suche verwendet keinerlei am Problem orientierte

Informationen über den Suchraum und testet erst am Ende (GT).

Heuristik

Wir können unsere Suche deutlich verbessern, wenn wir Wissen über das Problem in unsere Suche aufnehmen, den Suchraum also mit Heuristik einschränken. Ein einfacher erster Ansatz ist das Wissen, dass man Positionen, die man vorher schon versucht hat, nicht noch einmal versuchen muss. Wenn wir hier also nur noch den Abschnitt des Spielfeldes nach dem aktuellen Feld verwenden, können wir sicher keine Lösung übersehen. Diesen Weg geht ein verbessertes Programm.

Besser ist noch nicht gut

Dass dies tatsächlich zu einer Verbesserung führt, kann man bemerken, wenn man jeweils mit 6 × 6 Feldern arbeitet, bei 8 × 8 dauert es immer noch zu lange. Auch hier wieder die Frage: Warum ist das so?

Für eine Antwort berechnen wir einmal die Zahl der möglichen Belegungen, die bei einer vollständigen Suche identisch wäre mit der Zahl der untersuchten Möglichkeiten und dafür gilt bei ersten ganz unintelligenten Variante:

Bei der ersten Damen haben wir 8 × 8 = 64 Möglichkeiten, bei der zweiten wieder usw.

Das führt auf die Zahl von 64^8 Möglichkeiten, das sind etwa $2{,}8 \times 10^{14}$ Möglichkeiten. Eine kleine Abschätzung der Rechenzeit erklärt unser Warten auf eine Lösung. Bei der zweiten Variante, bei der wir einerseits Wiederholungen von Positionen ausschließen und andererseits die Fälle vermeiden, bei denen man nur eine andere Reihenfolge hat, sind es dann im Prinzip $\binom{64}{8} = \frac{64 \cdot 63 \cdot 62 \cdot 61 \cdot 60 \cdot 59 \cdot 58 \cdot 57}{8 \cdot 7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1}$ Möglichkeiten[1], auch noch eine sehr große Zahl!

HGT

Besser wird das Ganze nur, wenn wir uns umentscheiden und nicht erst eine mögliche Lösung voll generieren, sondern schon während des Hinzufügens weiterer Damen jeweils testen, ob die dadurch entstehende Konfiguration zulässig sein kann. Ein solches Vorgehen nennt man hierarchisches Generate and Test. Das so konstruierte Programm

```
def suche(alternativen, belegt):
    if len(belegt)==maxZahl:
        if zulaessig(belegt, belegt[1:]):
            return belegt
        return False
    if len(alternativen)==0:
        return False
    if not zulaessig(belegt, belegt[1:]):
        return False
    versuch = suche(alternativen[1:], [alternativen[0]]+belegt)
    if versuch:
        return versuch
    else:
        return suche(alternativen[1:], belegt)
```

1 Das ist die Zahl der Möglichkeiten, 8 Felder aus den 64 ohne Berücksichtigung der Reihenfolge auszuwählen.

findet schon nach kurzer Zeit eine Lösung.

Eine wichtige Verbesserung der uninformierten Suchverfahren erhält man also, wenn man Heuristiken in die Lösung integriert. Wir haben in diesem Fall – wie wir schon z.T. gesehen haben – einiges an elementarem Wissen über die Art der Lösungen, von dem wir noch nicht einmal alles verarbeitet haben:

- Es kann sich kein Feld wiederholen (s.o.) .
- Es kann sich keine Zeile wiederholen.
- Es kann sich keine Spalte wiederholen.

Dies führt zu einer naheliegenden Änderung bei der verwendeten Datenstruktur. Wir verwenden einfach eine Liste von Spaltenpositionen, in der die Positionen in dieser Liste selbst für die Zeilen stehen. Die Liste

[1, 3, 5, 8, 4, 7, 2, 6]

hat also denselben Informationsgehalt wie die Liste

[[1 , 1] [2 , 3] [3 , 5] [4 . 8] [5 , 4] [6 , 7] [7 , 2] [8 , 6]] .

Wenn die Zeilenpositionen sich nicht wiederholen können und sich von 1..8 abzählen lassen, dann brauchen wir sie nicht gesondert zu speichern. Das Testen auf gleiche Spaltenpositionen wird so auch erheblich vereinfacht, die „enthalten“-Funktion `in` erledigt die Frage.

Wir haben nun eine erste starke Verbesserung im Sinne des hierarchischen Generate and Test erreicht[1], da jetzt nur noch solche Nachfolger generiert werden, die prinzipiell erfolgversprechend sind, in diesem Fall also solche, die verschiedene Zeilenindizes und verschiedene Spaltenindizes haben.

Filtert man nun auch noch die heraus, welche die Diagonalenbedingungen verletzen, dann hat man das vorgegebene Suchverfahren im Sinne des hierarchischen Generate and Test optimiert.

1 Dabei ist fast belanglos, ob zunächst auch alle Möglichkeiten von Spaltenindizes erzeugt werden und dann die heraus gefiltert werden, die unzulässig sind, oder ob das schon in der Nachfolgerfunktion gewährleistet wird.