

## NP-vollständige Probleme

- Keine polynomiellen Algorithmen, falls  $P \neq NP$ .
- Viele wichtige Probleme sind  $NP$ -vollständig.
- Irgendwie **müssen** sie gelöst werden.
- Es reicht nicht zu sagen, daß das Problem  $NP$ -hart ist und daher nicht gelöst werden kann!

## Einführendes Beispiel

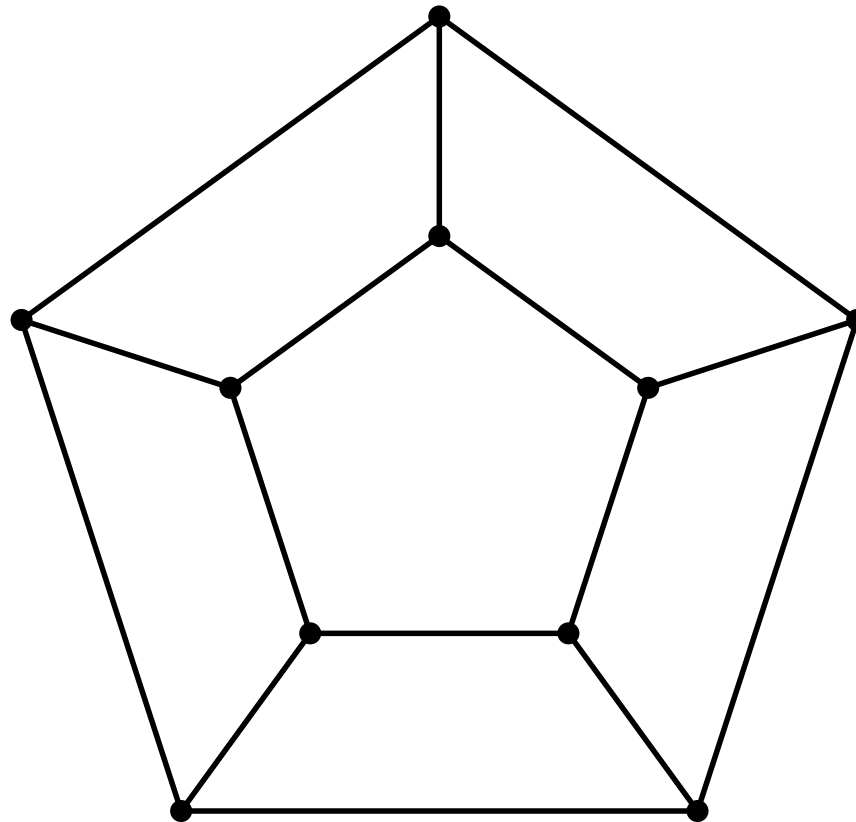
Gegeben: Ein Graph  $G = (V, E)$ .

Gesucht: Ein minimales *Vertex Cover*  $C \subseteq E$ .

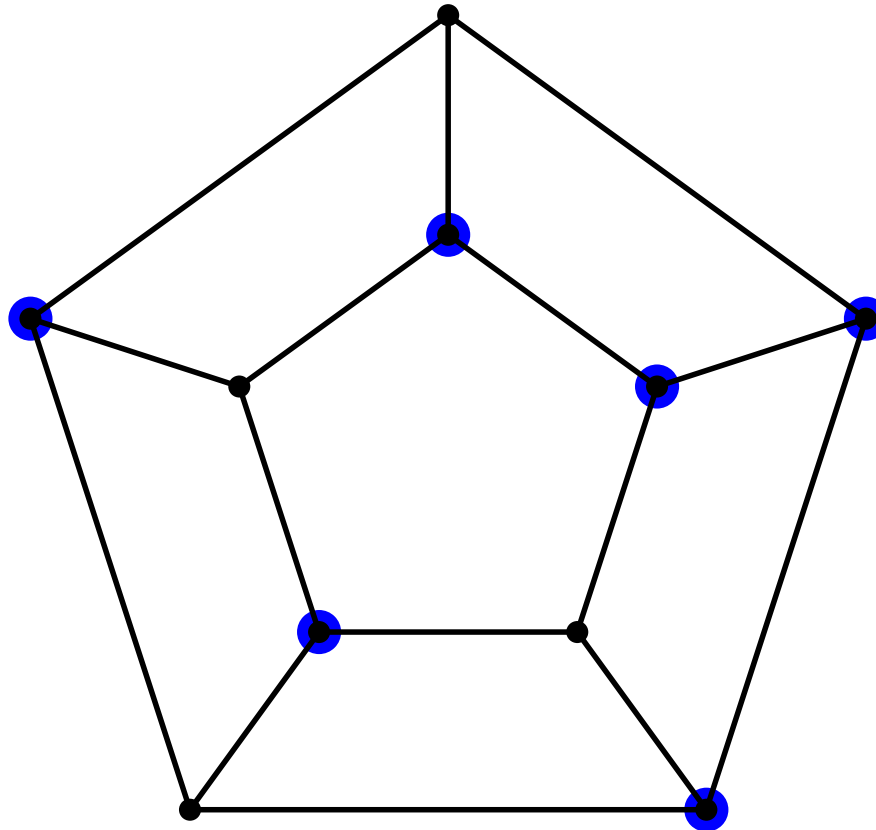
### Definition

Ein Menge  $C \subseteq V$  ist ein *Vertex Cover* von  $G = (V, E)$ , falls mindestens ein Endpunkt jeder Kante in  $E$  in  $C$  enthalten ist.

Beispiel



Beispiel



## Modellierung als ILP

Gegeben ist  $G = (V, E)$  mit  $V = \{v_1, \dots, v_n\}$ .

Minimiere  $v_1 + \dots + v_n$

unter  $0 \leq v_i \leq 1$  für  $i = 1, \dots, n$

$v_i + v_j \geq 1$  für  $\{v_i, v_j\} \in E$

$v_i \in \mathbf{Z}$  für  $i = 1, \dots, n$

**Jedes** *NP*-vollständige Problem läßt sich auf ein ILP zurückführen  
(aber meist ist das keine gute Idee).

## British Museum Method

Wichtige  $NP$ -vollständige Probleme sind oft **Suchprobleme**. In einem (sehr großen) Suchraum verbergen sich die Lösungen.

Eine mögliche Lösungsstrategie ist es daher, den Lösungsraum **vollständig** zu durchsuchen.

Für Vertex Cover bedeutet dies, alle  $C \subseteq V$  durchzuprobieren.

Das sind  $2^{|V|}$  Möglichkeiten.

Die Laufzeit beträgt  $O(|E|2^{|V|})$ .

## Backtracking

Wir betrachten einen Knoten  $v \in V$ .

Es gibt zwei Möglichkeiten  $v \in C$  oder  $v \notin C$ .

Falls  $v \notin C$ , dann  $N(v) \subseteq C$ , denn die zu inzidenten Kanten müssen ja abgedeckt werden.

(Mit  $N(v)$  bezeichnen wir die zu  $v$  adjazenten Knoten.)

Aus diesen Beobachtungen ergibt sich ein einfacher Algorithmus.

## Backtracking

Eingabe:  $G = (V, E)$

Ausgabe: Optimales Vertex Cover  $VC(G)$

$G_1 := (V - \{v\}, \{e \in E \mid v \notin e\})$

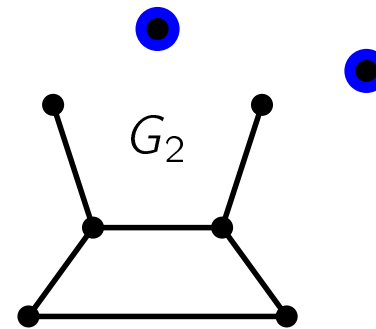
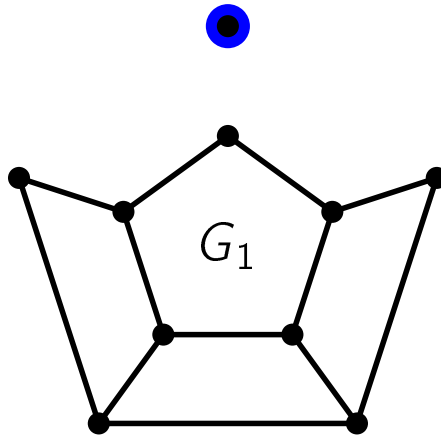
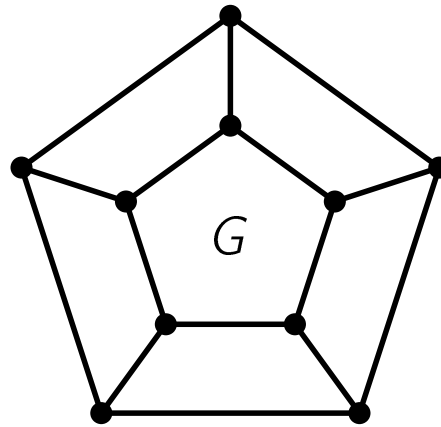
$G_2 := (V - \{v\} - N(v), \{e \in E \mid e \cap N(v) = \emptyset\})$

**if**  $|\{v\} \cup VC(G_1)| \leq |N(v) \cup VC(G_2)|$

**then return**  $\{v\} \cup VC(G_1)$

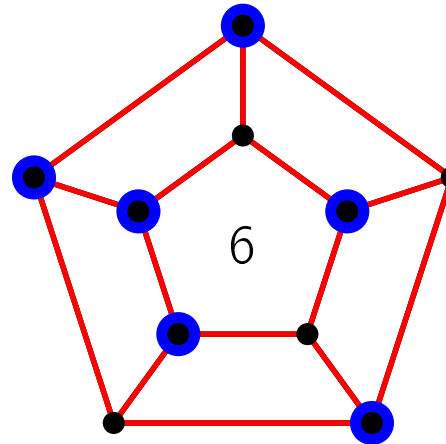
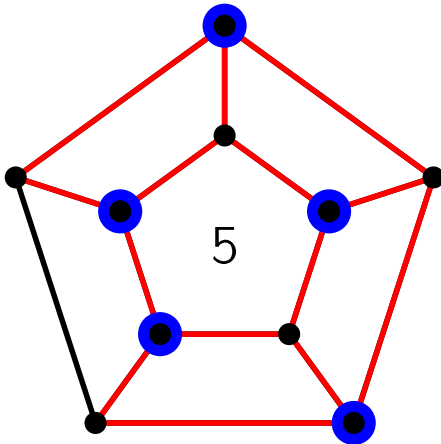
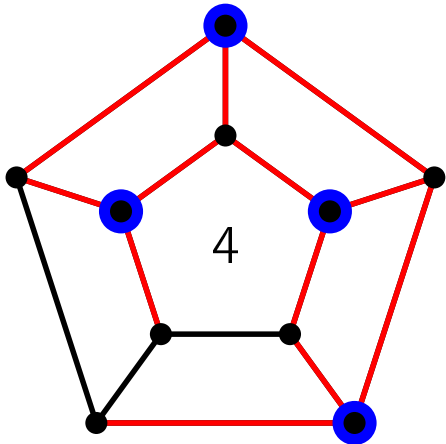
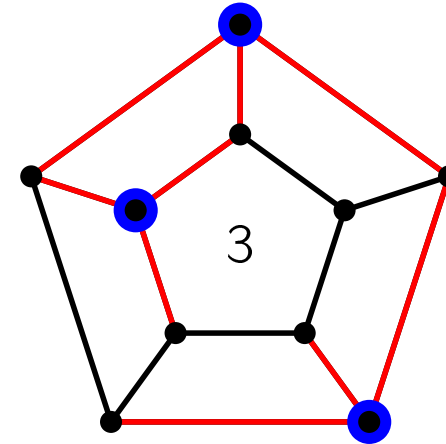
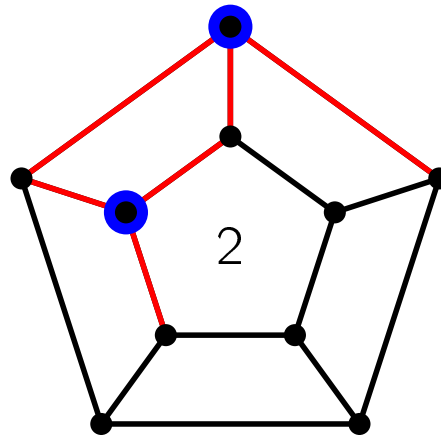
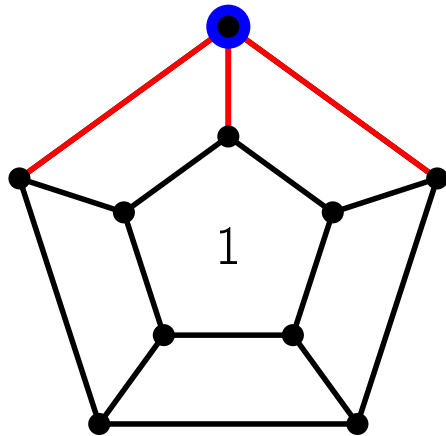
**else return**  $N(v) \cup VC(G_2)$



**Backtracking**

## Heuristiken

Zum Beispiel *Greedy*.



Wähle immer einen Knoten mit **maximalem** Grad.

## Approximationsalgorithmen

Jede Kante muß von **mindestens** einem ihrer Knoten abgedeckt werden.

Problem: Von **welcher**?

Lösung: Wir nehmen **beide**.

- Natürlich gibt es so keine Garantie, daß wir eine optimale Lösung finden.
- Das so gefundene Vertex Cover kann aber höchstens doppelt so groß sein wie ein optimales.

## Approximationsalgorithmen

Der Algorithmus könnte so aussehen:

```
 $C := \emptyset;$   
while  $E \neq \emptyset$  do  
  choose some  $e \in E$ ;  
   $V := V - e$ ;  
   $C := C \cup e$ ;  
   $E := \{e' \in E \mid e \cap e' = \emptyset\}$   
od;  
return  $C$ 
```

## Backtracking

Der Suchraum bei einem *NP*-vollständigen Problem, läßt sich oft so schreiben:

$$S_1 \times S_2 \times \cdots \times S_n$$

Jeder Punkt im Suchraum ist dann ein Vektor

$$(x_1, x_2, \dots, x_n)$$

mit  $x_i \in S_i$ .

Die Grundidee ist es, nacheinander  $x_1, x_2, \dots, x_n$  zu wählen.

## Bounding Functions

Sind bereits  $x_1, \dots, x_m$  gewählt, können wir manchmal bereits leicht entscheiden, daß es keine Lösung geben kann, die so beginnt.

Wir nehmen an, es gibt Prädikate  $B_1, \dots, B_n$ , so daß wenn

$$B_k(x_1, x_2, \dots, x_k)$$

**falsch** ist, dann gibt es keine Lösung  $(y_1, \dots, y_n)$  mit  $y_i = x_i$  für  $i = 1, \dots, k$ .

Der Algorithmus kann diese Prädikate verwenden, um den Suchraum einzuschränken.

## Beispiel

Beim 8-Damenproblem können wir als Suchraum

$$\{1, \dots, 8\}^8$$

wählen, denn in jeder Zeile steht eine Dame und sie muß in einer der 8 Spalten sein.

Wir setzen  $B_k(x_1, \dots, x_k) = \text{false}$ , wenn sich bereits zwei der ersten  $k$  Damen in den ersten  $k$  Zeilen auf den Positionen  $x_1, \dots, x_k$  bedrohen.

## Rekursiver Algorithmus

```
Backtrack( $k$ ) :  
for each  $x_k \in S_k$  do  
  if  $B_k(x_1, \dots, x_k)$  then  
    if  $(x_1, \dots, x_k)$  ist eine Lösung  
    then write  $(x_1, \dots, x_k)$  fi  
    if  $k < n$  then Backtrack( $k+1$ ) fi  
  fi  
od
```

Die Variablen  $x_1, \dots, x_n$  sind global.

Der Algorithmus muß mit *Backtrack*(1) aufgerufen werden.



## Bounding Functions

Wie wählen wir die Prädikate  $B_k$ ?

Ein Extremfall ist

$$B_k(x_1, \dots, x_k) = true$$

für alle  $x_i$ . Hier wird der Suchbaum niemals abgeschnitten und der gesamte Suchraum durchsucht.

Der andere Extremfall ist

$$B_k(x_1, \dots, x_k) = \begin{cases} true & \text{falls eine Lösung mit } x_1, \dots, x_k \text{ beginnt,} \\ false & \text{sonst.} \end{cases}$$

Hier werden alle Teilbäume abgeschnitten, die keine Lösung enthalten.

Beide Extremfälle sind im ungünstig. Entweder wird zuwenig abgeschnitten oder der Berechnungsaufwand für  $B_k$  wird zu groß.

## Iterativer Algorithmus

*Backtrack* :

$k := 1; R_k := S_k$

**while**  $k > 0$  **do**

**if**  $\{x \in R_k \mid B_k(x_1, \dots, x_{k-1}, x)\} \neq \emptyset$  **then**

**choose**  $x_k \in \{x \in R_k \mid B_k(x_1, \dots, x_{k-1}, x)\}$

$R_k := R_k - \{x_k\};$

**if**  $(x_1, \dots, x_k)$  ist eine Lösung **then write**  $x_1, \dots, x_k;$

$k := k + 1; R_k := S_k$

**else**  $k := k - 1$

**od**

## Abschätzung der Suchbaumgröße

Die Größe des (abgeschnittenen) Suchbaums kann grob abgeschätzt werden, **bevor** der Algorithmus abläuft.

Falls sich der Suchbaum im Level  $i$  um  $m_i$  verzweigt, dann ist die Anzahl der Blätter im Suchbaum etwa

$$\prod_{i=1}^n m_i.$$

Die Verzweigungsgrade  $m_i$  lassen sich leicht für einen **zufälligen** Pfad im Suchbaum leicht bestimmen.

Experimente zeigen, daß die Schätzungen im allgemeinen ganz gut sind.

Es können mehrere Schätzungen kombiniert werden (aber nicht zu viele).

## Abschätzung der Suchbaumgröße

*Estimate:*

$k := 1; m := 1; r := 1;$

**repeat forever do**

$T_k := \{ x_k \in S_k \mid B_k(x_1, \dots, x_k) \}$

**if**  $|T_k| = 0$  **then return**  $m$ ;

$r := r \cdot |T_k|;$

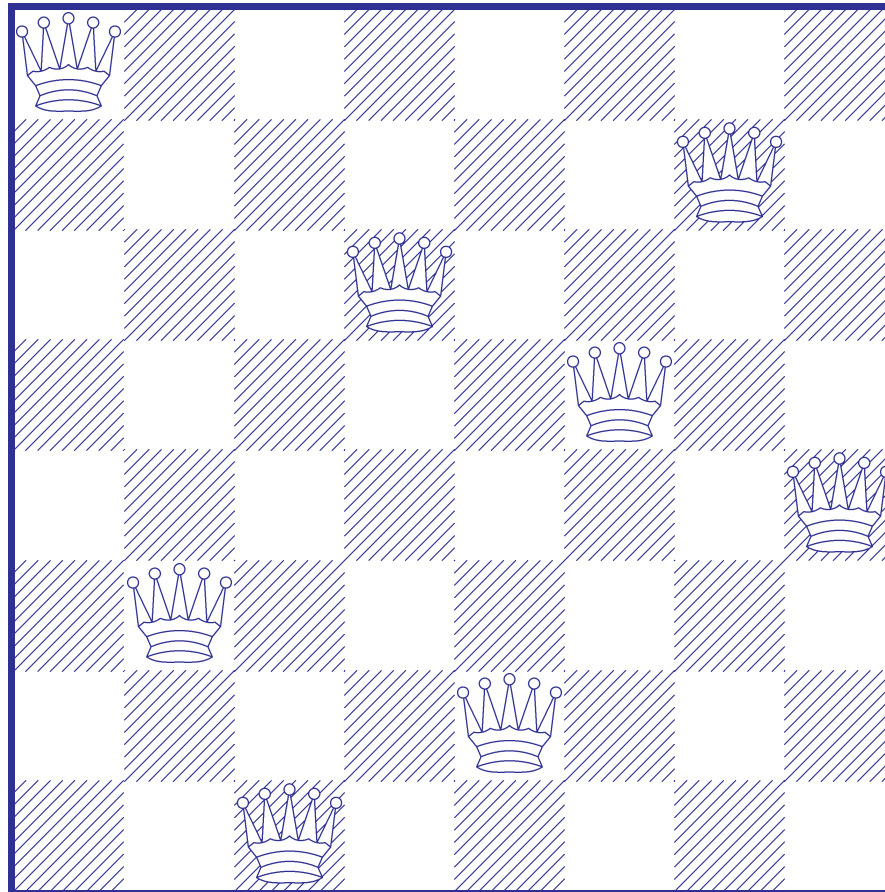
$m := m + r;$

$k := k + 1;$

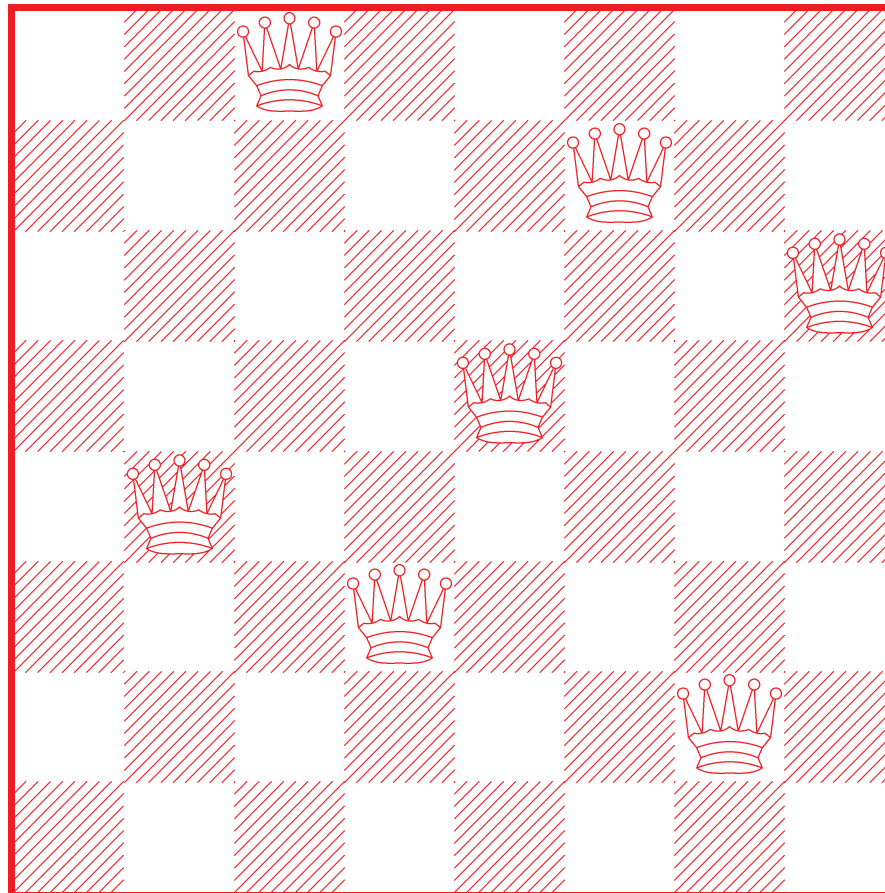
$x_k :=$  zufälliges Element  $\in T_k$

**od**

Dieser Algorithmus schätzt die Suchbaumgröße der rekursiven oder iterativen Backtrackalgorithmen.

**Beispiel**

Das 8-Damen-Problem



Wir verwenden den Abschätzungsalgorithmus. Es gibt jeweils 8, 5, 3, 3, 2, 1, 1 Möglichkeiten, nächste Dame zu setzen. Das ergibt geschätzte **3529** Positionen im Suchbaum.

Ein Programm für das 8-Damen-Problem mit Backtracking:

```
#define N 8
int x[N], c[N], d1[2 * N - 1], d2[2 * N - 1];

int main(void)
{
    int k = 0, i;
    while(k >= 0) {
        i = x[k];
        if(i == N) k--, i = x[k]++, c[i] = d1[k - i + N - 1] = d2[i + k] = 0;
        else if(!c[i]&&!d1[k - i + N - 1]&&!d2[i + k])
            if(k == N - 1) {
                for(i = 0; i <= k; i++) printf("%d ", x[i]);
                printf("\n");
                x[k]++;
            }
            else c[i] = d1[k - i + N - 1] = d2[i + k] = 1, k++, x[k] = 0;
        else x[k]++;
    }
    return 0;
}
```

Ist das Programm korrekt?

## Abschätzung der Suchbaumgröße

Der Schätzwert für die Suchbaumgröße war **3529**.

Lassen wir das Programm laufen, ergibt sich, daß **1830** gültige Positionen getestet werden.

Insgesamt gibt es 92 Lösungen.



Folgendes Programm schätzt die Suchbaumgröße automatisch:

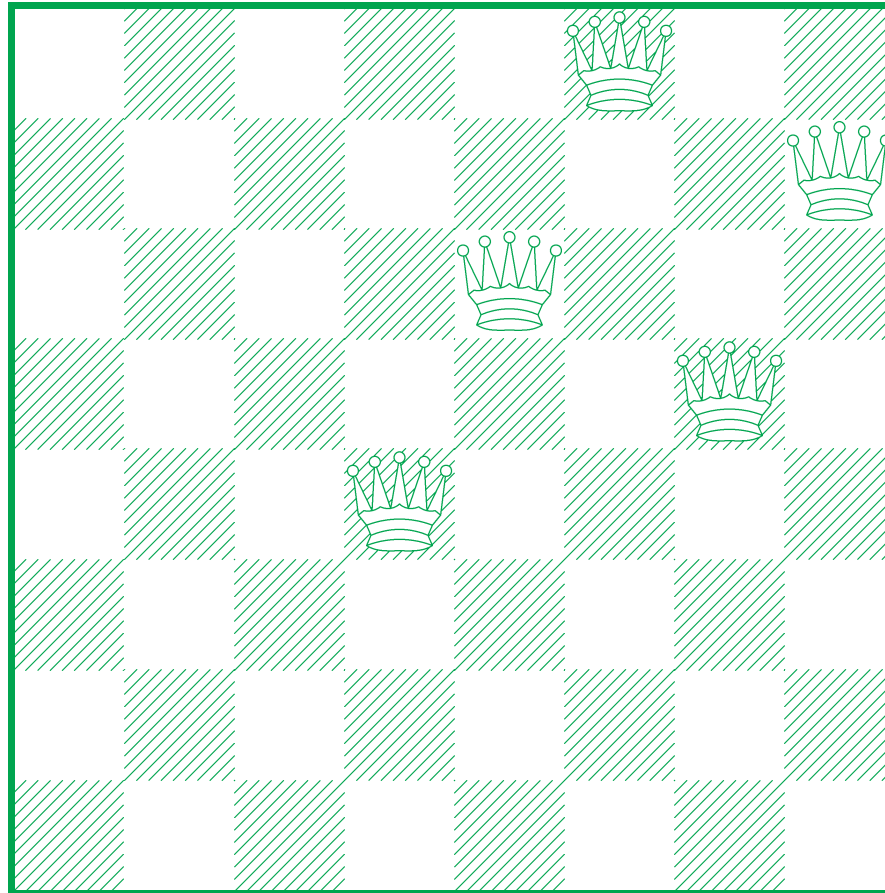
```
#define N 8
int c[N], d1[2 * N - 1], d2[2 * N - 1], t[N];

int main(void)
{
    int k = 0, i, z, r = 1, m = 1;
    while(1) {
        z = 0;
        for(i = 0; i < N; i++)
            if(!c[i]&&!d1[k - i + N - 1]&&!d2[i + k]) t[z] = i, z++;
        if(z == 0) break;
        m += r *= z;
        i = t[random() % z];
        printf("%2d %2d %d\n", i, z, m);
        c[i] = d1[k - i + N - 1] = d2[i + k] = 1;
        k++;
    }
    return 0;
}
```

Lassen wir dieses Programm laufen, ergibt sich folgende Ausgabe:

```
5  8  9
7  5 49
4  4 209
6  3 689
3  2 1649
```

Die mit Hand geschätzte Zahl war **1609** und die tatsächliche Suchbaumgröße **1830**.



Dies ist die geratene Belegung des Programms.

Für  $N = 16$  ergibt sich diese Ausgabe:

13 16 17

1 13 225

10 10 2305

4 10 23105

15 7 168705

0 5 896705

12 7 5992705

9 3 21280705

6 3 67144705

8 3 204736705

14 1 342328705