Available online at www.sciencedirect.com

ScienceDirect

Discrete Mathematics 307 (2007) 2535–2551

DISCRETE MATHEMATICS

www.elsevier.com/locate/disc

A group-based search for solutions of the n-queens problem

Matthias R. Engelhardt

Unterreichenbacherstr. 9, 90455 Nuernberg, Germany

Received 9 July 2005; received in revised form 13 October 2006; accepted 18 January 2007
Available online 4 February 2007

Abstract

The n-queens problem is a well-known problem in mathematics, yet a full search for n-queens solutions has been tackled until now using only simple algorithms (with the exception of the Rivin–Zabih algorithm). In this article, we discuss optimizations that mainly rely on group actions on the set of n-queens solutions. Most of our arguments deal with the case of toroidal queens; at the end, the application to the regular n-queens problem is discussed, and also the Rivin–Zabih algorithm.
© 2007 Elsevier B.V. All rights reserved.

Keywords: n-Queens problem; Finite group action; Complete enumeration; Backtracking algorithm

1. Introduction

The search for solutions to the n-queens problem, i.e. the placing of n queens on an $n \times n$ chessboard so that none attacks any other, is well known. The availability of computers since about 1960 has given it a new impetus. Nowadays the n-queens problem is often presented in computer science courses and there it frequently serves as an exercise.

However, an astonishing aspect of the methods used currently for a complete search is that—as far as I know, and with the exception of the Rivin–Zabih algorithm—they are not very advanced. The n-queens problem is seen rather as an example which shows that backtracking algorithms are of little help in problems with exponential growth. In 1999 the numbers of solutions were only known for boards up to 23×23, both for the regular n-queens problem and for the variant with the board wrapped round a torus. Additionally, I know of no attempt to enumerate the solutions.

It seems that the large number of solutions is a deterrent to any enumeration of them. There are two possible approaches to dealing with large sets: one way is to consider the elements of the solution set as chance results; but those are not so interesting and one is only concerned with calculating the total number, either exactly or asymptotically, or with the frequency with which they appear in a larger set. This is the method presently used in solutions to the n-queens problem. The other possibility is to classify the elements of a solution set and to find relationships between them. Group actions are an especially potent tool here. This paper describes the first steps along this way.

The words ‘not so interesting’ should not be taken literally. Indeed, there are about 30 entries in the bibliography of the n-queens problem of Prof. Kosters (http://www.liacs.nl/∼.kosters/nqueens.html), and the work in that direction has been fruitful in the study of constraint-satisfaction problems. Work in this direction is probably even more interesting for similar real-world applications than this article. Yet, while the subject is the same, the perspective is quite different.

E-mail address: matthias.r.engelhardt@web.de.

0012-365X/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.disc.2007.01.007

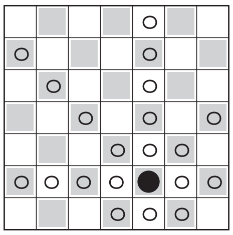

Fig. 1. A queen and the attacked cells in the regular variant.

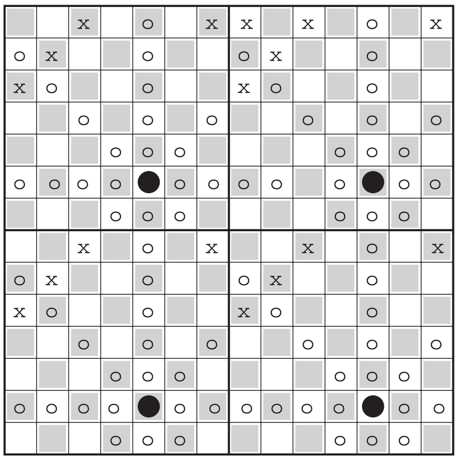

Fig. 2. A queen and the attacked cells in the torus variant. Cells that are threatened by torus queens only are marked with ‘x’. In this way, the completion of the diagonal becomes evident.

On the one hand, I want to show in this article how the search can be accelerated considerably by simple programming improvements and, on the other hand, how one can make much more use of symmetries, particularly in the torus variant of the problem. Using these methods I have been able to calculate the next solution sets for the torus problem with only moderate computing power: T_{23} was already known (almost 129×10^6), new results are T_{25} (almost 2×10^9), T_{29} (almost 606×10^9) and most recently also T_{31} (just over 13.4×10^{12}).

My method saves the solution sets in a format which allows the enumeration and investigation of the individual properties of the solutions. It saves a complete transversal, i.e. one solution for each orbit, under action of the largest possible group.

The aspect of symmetry was already used in a weak sense in the classical case, the normal queens problem on an 8×8 board; here there are 92 solutions and one usually enumerates 12 of them, from which the others are generated by rotation and mirroring, i.e. the action of the dihedral group $\mathbf{D}_4$. The result, which reduces these 12 cases through action of larger groups (congruence, similarity) to 6 and then to 4, seems to be new. The fact is also hardly mentioned that for the torus variant and $n < 13$ there is only one trivial solution using the action of the affine group.

To be more specific we come to an informal definition of the problem. The n-queens problem has two major variants: the *regular n-queens problem*, and the *toroidal n-queens problem*, also called the *modular n-queens problem.* For the definition of both variants, we cite Rivin et al. [2], the article that summarizes the basic facts on the n-queens problem. We start with the regular variant:

> The n-queens problem asks how many ways can one put n queens on an $n \times n$ chessboard so that no two queens attack each other. In other words, how many points can be placed on an $n \times n$ grid so that no two are on the same row, column or diagonal (see Fig. 1).

And for the toroidal n-queens problem:

> . . . a more tractable problems seems to be the *toroidal n-queens problem*: How many ways can one place n queens on an $n \times n$ chessboard so that no two queens can be on the same row, column or extended diagonal.

The term 'extended diagonal' needs some explanation. Two ideas may be helpful in understanding it. The first is to repeat the chessboard endlessly in the plane, with period n for both rows and columns, as is shown in Fig. 2.

The second idea is to wrap the chessboard on a cylinder, by identifying the left and right borders, and then to form a torus from the cylinder, identifying top and bottom borders.

The rest of this section gives formal definitions of sets, groups and mappings used in the paper and ends with a formal definition of the full sets of solutions ($\mathbf{Q}_n$ for the regular n-queens problem and $\mathbf{T}_n$ for the toroidal n-queens problem).

The simple algorithm used until now is given at the beginning of Section 2, and improvements to it are described step by step in the rest of Section 2. The first two (Sections 2.2 and 2.3) are just programming techniques and mainly of interest in computer science; the other sections extend symmetry arguments and are interesting from a mathematical point of view.

Most of the improvements are concerned with the toroidal problem. Section 3 shows in more detail how this was applied to the computation of the set of torus solutions for $n=29(\mathbf{T}_{29})$. At the end, the results are given in some detail, including results for the performance of the various improvements.

The alternative approach of Rivin and Zabih, which does not use backtracking, is discussed in Section 4.

Section 5 discusses some modifications that are necessary to handle the regular n-queens problem with the ideas developed for the torus variant.

1.1. Sets and groups of interest

The main idea of the improvements is to use group actions. That means that some groups *act* on some sets. Here we introduce these sets and groups.

Starting point for that is the chessboard; we take $\mathbf{F}_n := \mathbb{Z}_n \times \mathbb{Z}_n$ as the set of cells of the chessboard; $\mathbb{Z}_n$ means $\{0, \ldots, n-1\}$ in this paper, either as set or as group or as ring, depending on the context.

Important subsets of $\mathbf{F}_n$ are the diagonals. We distinguish diagonals to the left and to the right, calling them either diagonals or anti-diagonals. The numbering is different for the regular and the torus variant. To keep it general, we use numbers forming a ring U for numbering these (anti-)diagonals. Depending on whether we are handling the regular n-queens problem or the toroidal, we use for U either the ring of integers $\mathbb{Z}$ or the ring of integers modulo n, $\mathbb{Z}_n$. Taking U with its ring structure means that addition and subtraction also depend on U, i.e. in the torus variant, sums, differences, and products are taken mod n.

Further, we use four projections p_r, p_c, p_d and p_a from $\mathbb{Z}_n \times \mathbb{Z}_n$ to U which map a cell of the board to its row, its column, diagonal or anti-diagonal, respectively.

$$p_\mathrm{r}(x, y) = y, \tag{1}$$

$$p_\mathrm{c}(x, y) = x, \tag{2}$$

$$p_\mathrm{d}(x, y) = x - y, \tag{3}$$

$$p_\mathrm{a}(x, y) = x + y. \tag{4}$$

The groups of interest consist of movements which map the chessboard onto itself. These movements must be reversible, i.e. they are bijective mappings $\mathbf{F}_n \to \mathbf{F}_n$.

Groups $\mathbf{M}$ *and* $\mathbf{D}_4$: The most simple nontrivial group $\mathbf{M}$ consists of the identity map and the vertical reflection S only, being isomorphic to $\mathbb{Z}_2$. S is defined by $(x, y) \mapsto (x, n-1-y)$.

Next, we include the 90° rotation in the group. Doing that, we get the horizontal reflection and also reflections at the (anti-)diagonal. That gives the dihedral group $\mathbf{D}_4$. That is the group of congruent mappings within the given border, i.e. without shifts on the torus. This group has eight elements, and is generated by the two elements R (rotation by 90°) and S (vertical reflection) with the relations $R^4 = 1$, $S^2 = 1$, and $S \circ R = R^3 \circ S$.

Group $\mathbf{D}_{4,n}$: For the torus problem, we can start with the group of all shifts i.e. $\mathbb{Z}_n \times \mathbb{Z}_n$ in mathematical notation. It is an abelian group, generated by the horizontal shift H and the vertical shift V with the relations $H^n = 1$, $V^n = 1$, and $H \circ V = V \circ H$.

It is more natural, however, to go to a group containing both previous groups. This way, we come to the group $\mathbf{D}_{4,n}$ of all congruent mappings on the torus. The new group has $\mathbf{D}_4 \times \mathbb{Z}_n \times \mathbb{Z}_n$ as underlying set but it is not the product group. Instead, we have the additional relations $S \circ H = H^{-1} \circ S$, $S \circ V = V \circ S$, $R \circ H = V \circ R$, and $R \circ V = H^{-1} \circ R$.

An element of $\mathbf{D}_{4,n}$ is given by a triple $(t_{\mathrm{h}}, t_{\mathrm{v}}, d)$ where t_{h} and $t_{\mathrm{v}} \in \mathbb{Z}_n$ are the horizontal and vertical translation, respectively, and d is some element from $\mathbf{D}_4$, i.e. a rotation or reflection.

Group $\mathbf{Sim}_n$: Further, we can add *central extensions* c_a to the movements

$$c_a\colon (x, y) \mapsto (ax, ay). \tag{5}$$

We can use c_a if their factor a is a unity in the ring $\mathbb{Z}_n$. That is equivalent to $\gcd(a, n) = 1$. If the factor were no unity, the mapping would not be bijective.

There is a group which contains only these central extensions. The composition $c_a \circ c_b$ gives the central extension c_{ab}. More interesting is the group generated from these central extensions and shifts, rotations and reflections. In this way, the congruent motions are generalized to similar motions. We call that group $\mathbf{Sim}_n$.

An element of $\mathbf{Sim}_n$ is given by a quadruple $(a, t_{\mathrm{h}}, t_{\mathrm{v}}, d)$ where a is the expansion factor, $1 \leqslant a < n/2$, $\gcd(a, n) = 1$, and the other parts are as above translation components, and rotation or reflection.

Group $\mathbf{Aff}_n$: Finally, the largest group that we consider in this article is the group of all affine bijective mappings of $\mathbf{F}_n$ which we call $\mathbf{Aff}_n$.

There is a chain of inclusion of these groups:

$$\mathbf{M} \subset \mathbf{D}_4 \subset \mathbf{D}_{4,n} \subset \mathbf{Sim}_n \subset \mathbf{Aff}_n. \tag{6}$$

The two groups of torus shifts (without rotations and reflections) and of central expansions do not fit into this chain; however, they are contained in $\mathbf{D}_{4,n}$ and $\mathbf{Sim}_n$, respectively.

Note that all motions, i.e. all elements of these groups, can be described by 3×3 matrices if we append to each cell of $\mathbf{F}_n = \mathbb{Z}_n \times \mathbb{Z}_n$ a third coordinate which always has value 1. Elements of the last and largest group, $\mathbf{Aff}_n$, are described by a matrix

$$\begin{pmatrix} a_{11} & a_{12} & t_1 \\ a_{21} & a_{22} & t_2 \\ 0 & 0 & 1 \end{pmatrix}, \tag{7}$$

where a_{ik} and t_i are in $\mathbb{Z}_n$, and the determinant $a_{11}a_{22} - a_{21}a_{12}$ is a unity in $\mathbb{Z}_n$. t_1 and t_2 describe the translation part of the affine motion.

Having defined the chessboard and the groups acting on it, we come to *derived actions*. For that, we consider—more generally—subsets of $\mathbf{F}_n = \mathbb{Z}_n \times \mathbb{Z}_n$. For instance, solutions for the n-queens problem are subsets of $\mathbf{F}_n$.

To every action of the above groups on $\mathbf{F}_n$, there is also a natural action on the power set $\mathfrak{P}(\mathbf{F}_n)$ (see, for instance, [1], 2.1.10 in Section 2.1, or at the beginning of Section 5.3). In that action, a motion α maps the set of cells $\{f_i \mid i = 0 \dots k-1\}$ to $\{\alpha(f_i) \mid i = 0 \dots k-1\}$.

We denote by $\mathfrak{P}_k(\mathbf{F}_n)$ the set of subsets of $\mathbf{F}_n$ having k elements. So, we have a decomposition

$$\mathfrak{P}(\mathbf{F}_n) = \bigcup_{0 \leqslant k \leqslant n^2} \mathfrak{P}_k(\mathbf{F}_n) \tag{8}$$

(k lies in the range 0 to n^2 since $\mathbf{F}_n$ has n^2 elements).

The groups of movements given above act also on every $\mathfrak{P}_k(\mathbf{F}_n)$. To see that, we need only check that a subset of k elements is mapped by a motion α to a subset of k elements; that is obvious as all α are bijective.

The n-queens solutions lie in $\mathfrak{P}_n(\mathbf{F}_n)$. Mostly, we are interested in a smaller subset of it. That subset corresponds to the 'n-rook-solutions', and is given by the graphs of all permutations. Usually, the set of permutations of n elements is given by the symmetric group S_n. At this point, we emphasize that in all the following text the symmetric group is taken only as a set, not with its group structure. We use actions of the above groups on the set S_n but that should not be confused with group operations in S_n.

Formally, we have the imbedding which maps every permutation onto its graph

$$S_n \to \mathfrak{P}_n(\mathbf{F}_n), \quad p \mapsto \{(i, p(i)) \mid 0 \leqslant i < n\}. \tag{9}$$

We denote its image by $\overline{S_n}$.

It is easy to check that $\mathbf{M}$, $\mathbf{D}_4$, $\mathbf{D}_{4,n}$ and $\mathbf{Sim}_n$ act on $\overline{S_n}$ while $\mathbf{Aff}_n$ does not. A counterexample is the shear mapping $s:(x,y)\mapsto(x,x-y)$ which is in $\mathbf{Aff}_n$, and the identity id of S_n, for $n>1$. $\overline{\mathrm{id}}$ is the diagonal $\{(k,k)|0\leqslant k<n\}$, and its image under s is the zero row $\{(k,0)|0\leqslant k<n\}$ which is *not* in $\overline{S_n}$.

A few other subsets of $\mathfrak{P}(\mathbf{F}_n)$ or $\mathfrak{P}_n(\mathbf{F}_n)$ will be introduced as needed.

1.2. Formal definition of the problem

In short, both variants of the n-queens problem ask for a subset of the symmetric group S_n, for some given $n\in\mathbb{N}$. These subsets are called $\mathbf{Q}_n$ for the regular variant and $\mathbf{T}_n$ for the toroidal variant.

For the elements p of these subsets, we request the following additional property: both $p_{\mathrm{d}}\circ(\mathrm{id}\times p)$ and $p_{\mathrm{a}}\circ(\mathrm{id}\times p)$ are injective.

The search for these solutions will use group actions on $\overline{\mathbf{Q}_n}$ and $\overline{\mathbf{T}_n}$.

Again, it is easy to check that $\mathbf{M}$ and $\mathbf{D}_4$ act on both $\overline{\mathbf{Q}_n}$ and $\overline{\mathbf{T}_n}$. The groups $\mathbf{D}_{4,n}$ and $\mathbf{Sim}_n$ act on $\overline{\mathbf{T}_n}$, but not on $\overline{\mathbf{Q}_n}$; that is just the formal way of saying that a regular n-queens solutions cannot be shifted or expanded.

2. Improvement steps

This section is devoted to the algorithm for finding $\mathbf{Q}_n$ or $\mathbf{T}_n$.

The following point is a little unusual: as we describe the features and ideas of the author's program, we also give details of the syntax which activates these features. To give concrete examples, we refer to the syntax in the next section when describing how concepts were combined in the search for $\mathbf{T}_{29}$. The syntax is otherwise not of interest.

2.1. The basic algorithm

The basic algorithm is a standard example for backtracking, or even an exercise in many lectures in computer science. The author took it as a starting point, improving it step by step.

Describing the algorithm, we use the word 'free' in two meanings: We call a *cell* 'free' if it is not threatened by any queen already placed. For a *line* (row, column or (anti-)diagonal), 'free' means that no queen is placed on that line.

The basic algorithm places the queens on the board row by row, starting with an empty board; it consists of the three main steps:

Step 1: Generate a list of the free cells in the next row. If there are none, backtrack, i.e. go to step 2 of the previous row.

Step 2: Place a queen on the next free cell, and proceed with step 1 for the next row, if there is a next row. Otherwise, proceed to step 3. If the list of free cells is exhausted, backtrack, i.e. remove the queen on the previous row and continue with step 2 in the previous row.

Step 3: If you have queens on all rows, you have a solution; count it or store it, then backtrack.

There are many different versions of how this algorithm is implemented. Some of the publications on the n-queens problem deal with that point.

As stated above, some other publications study variants of the algorithm that find only one solution which is, in some sense, close to a given permutation or to a permutation generated at random. In this article, we concentrate on algorithms finding the *full* subsets $\mathbf{Q}_n$ and $\mathbf{T}_n$.

An important point in the implementation is how the actual state of the search is represented in the program. Coming from mathematical notation, the use of arrays is the common method. Some aspects of the algorithm are described that way in the following text.

However, the run time can often be improved by avoiding arrays with their indexing. Instead, we use data structures for every cell and for lines of interest, and we chain them. Especially, for backtracking, unlinking and relinking may be an elegant implementation.

These points are handled as implementation details and not explained further here; so, speaking of arrays in the following does not necessarily mean that arrays and indexing are used in the author's programs.

Also, it is not the focus of this article to compare CPU times needed or the effectiveness of different algorithms. The focus is to present ideas which allow computations that previously needed too much computing effort. Only in Section 3.4 are some figures given concerning the performance of the improvements.

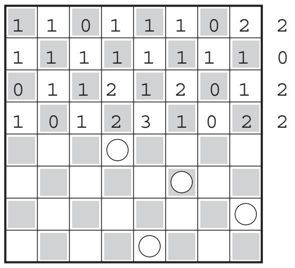

Fig. 3. A situation where the program should backtrack.

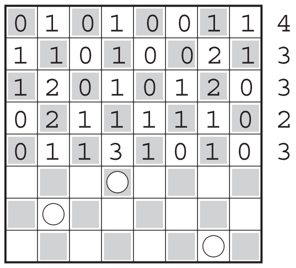

Fig. 4. A situation where the next line is not the best to continue.

2.2. *Further look ahead—not just next row*

Most programs just check the next row for the next queen, i.e. they check if there is a free cell in the next row. Many lectures probably recommend that in order to minimize the effort for a single step. That may well be correct for the n-queens problem with smaller chessboards, say up to 12×12 cells. However, for larger n, it is better to invest more work.

The author's program maintains an $n \times n$ array containing, for every cell, a count of how many queens threaten that cell. With that array, the program determines if, in any free row, there is no unthreatened cell left; if so, it backtracks.

To illustrate that: say, we placed queens on rows 1–4 on an 8×8 chessboard, threatening all cells of row 7 (see Fig. 3). So, row 7 is free while none of its cells are free. The program will then backtrack and move the queen in row 4; it will not try to place queens on row 5 and 6, although that may be possible.

A second improvement to the base algorithm: do not necessarily handle the rows in their natural order, but search for the row with the fewest free cells left. E.g. if we are handling row 3 and detect, as in Fig. 4, that in row 5 only two free cells are left while there are 3 or 4 in the rows 4, 6, 7 and 8, then we place the next queen in row 5.

In this way, we cut down the tree that must be traversed for the search.

With just this simple improvement, the author has made the search run for the 25×25 torus problem.

2.3. *Using columns and diagonals also*

All other programs that the author knows handle the board row by row. However, that is not necessary. It is only necessary to split the problem into cases in such a way that the program catches all possibilities. (Placing a queen in the next row effectively splits the rest of the problem into cases.) For that splitting of the problem, columns are as good as rows. For the torus problem, we can additionally use diagonals or anti-diagonals.

The order of processing of these split-cases may depend dynamically on the queens that are already on the board.

2.4. *Exploitation of symmetries*

The main improvement in the algorithm consists in the exploitation of symmetries. That means that the program searches only few solutions and can find or count the other solutions as symmetric images of them.

At this point, a historical remark on the use of symmetries in former programs seems in place. In some special cases, exploitation of symmetry has been implemented already. Yet most programs use only very basic symmetry by reflection. For the regular n-queens problem, they place the first queen only in the left half of the first line and eventually multiply the number found by two. For odd n, the case where the first queen is in the middle of the row must be counted separately. That means that these programs use the action of $\mathbf{M}$.

Schimke [4] went one step further in this direction. His program also recognizes a newly found solution as a rotated image of a solution that it found previously. That is a use of the action of $\mathbf{D}_4$.

Searching for torus solutions, we have to take into account shift operations too. That reduces the range of search significantly. But programs often use only horizontal translations, usually placing the first queen on the left, bottom corner (first row, first column), and multiplying the counted solutions by n to get the total number (see, for instance, [2]). That is a use of the cyclic group C_n which is not mentioned further in this article. So far for the historical remark.

For more intensive application of symmetry, we use a few notions of group theory, as given, e.g. in Kerber [1]. The n-queens problem is not mentioned in the book but the same ideas may be applied.

The symmetries used for the n-queens problem stem from the groups introduced in 1.1. We use the action of these groups on $\overline{\mathbf{Q}_n}$ and on $\overline{\mathbf{T}_n}$.

An n-queens solution (an element of $\overline{\mathbf{Q}_n}$ or $\overline{\mathbf{T}_n}$) is called *symmetric with respect to a motion* (an element of the group) if and only if it is mapped onto itself by the motion. Or in group theoretic language: if the motion is in the *stabilizer* of the solution.

An n-queens solutions p is called symmetric to another solution q if some motion maps p to q. All solutions symmetric to a given q form an *orbit*.

In this aspect, the set of all solutions is decomposed to the *orbits* of the group under discussion. The search for all solutions is reduced to finding a *transversal* of all solutions with respect to the group.

Sometimes we can use *normalization* or—with the same meaning—*canonization*; that means that we assign a special element to every orbit. The simplest canonization is to use the smallest element of every orbit for that, but other canonizations are possible as well, and may be easier to implement.

Considering these symmetries while searching makes the algorithm rather complex; we can avoid that if we save the solutions in a file and evaluate them later in other programs. In this way, the complexity is transferred to these other programs.

The idea for the search is to use further restrictions during the search, in order to get as few elements of each orbit as possible. For that, the author used a classification of $\mathbf{T}_n$ which is oriented on geometric aspects. Most of it can also be applied to the full set S_n.

That classification uses three properties of a permutation that are invariant under the action of $\mathbf{D}_{4,n}$: the *minimal neighbor distance*, the *length* of the *longest chain* of queens in that distance, called 'knight length' here, and the *minimal distance at the end* of such a chain, called 'secondary minimal distance'.

These three properties are defined formally in this subsection, and used in the following subsections.

The first property is the minimal distance on a neighbor row or column; for that, we have the following definition:

Definition 2.1. The *neighbor distance at i* of $p \in S_n$ is

$$\begin{aligned} d_{p,i} := \min(&\{|(p(i+k)-p(i))| \mid k=\pm 1\} \\ &\cup \{n-|(p(i+k)-p(i))| \mid k=\pm 1\} \\ &\cup \{|p^{-1}(p(i)+k)-i| \mid k=\pm 1\} \\ &\cup \{n-|p^{-1}(p(i)+k)-i| \mid k=\pm 1\}). \end{aligned} \tag{10}$$

The *minimal neighbor distance of $p \in S_n$*, written as d_p, is the minimum of $d_{p,i}$, for $0 \leqslant i < n$.

As all is finite, the minimum always exists. Note that this definition differs from the Manhattan distance: it is restricted to cells on adjacent rows or adjacent columns, and in this case, the neighbor distance is Manhattan distance minus one.

With that definition, we have two lemmata:

Lemma 2.2. *The minimal neighbor distance is a class function on* $\mathbf{T}_n$ *and on* S_n, *under the action of* $\mathbf{D}_{4,n}$. *Or formally*,

$$d_p = d_{\alpha(p)} \quad \text{for all } \alpha \in \mathbf{D}_{4,n}. \tag{11}$$

Proof. Before we start the proof, we should interpret the four lines of definition (10). The first two give the distance of queen $(i, p(i))$ to the queens on the adjacent columns. The first line is the distance within the interval $[0, n-1]$, and the second line is the distance over the border where 0 and n are identified. Lines three and four give the distance to queens on the adjacent rows in the same way.

It is sufficient to check formula (11) for the generators of the group $\mathbf{D}_{4,n}$. That means we must check it for translations, the 90° rotation R, and for the vertical reflection M. Let α be such a mapping. α maps both the permutation p, and the basic set $\mathbf{F}_n$. It maps the cell $(i, p(i))$ of (10) to some cell (x, y). We show that $d_{p,i} = d_{\alpha(p),x}$.

If α is a translation or vertical reflection, queens on adjacent rows are mapped to adjacent rows, and queens on adjacent columns to adjacent columns. If α is the 90° rotation, rows are mapped to columns and vice versa. So, a single term of (10) may change its position in the four lines of the formula, but it stays in the set over which the minimum is taken. Hence, $d_{p,i} = d_{\alpha(p),x}$.

To come from $d_{p,i}$ to d_p, we take the minimum over $i = 0 \ldots n-1$. For $d_{\alpha(p)}$, we must take the minimum over $x = 0 \ldots n-1$ where x is defined as above. Since $\alpha(p)$ is a permutation, that is true. The minimum is taken over the same set of numbers, whence $d_p = d_{\alpha(p)}$. □

Lemma 2.3. *The minimal neighbor distance* d_p *of a permutation* p *is* k *if and only if there is an* $\alpha \in \mathbf{D}_{4,n}$ *such that* $\alpha(p)(0) = 0$ *and* $\alpha(p)(1) = k$, *and there is no* $\alpha \in \mathbf{D}_{4,n}$ *such that* $\alpha(p)(0) = 0$ *and* $\alpha(p)(1) < k$.

Proof. We first prove the necessity of the two conditions. Let p be a permutation and let k be its minimal neighbor distance, i.e. $d_p = k$. We construct an appropriate α by composing some generator elements of $\mathbf{D}_{4,n}$. Due to the finiteness of all variables, there must be some i such that $d_p = d_{p,i}$. That means that the queen at $(i, p(i))$ has some neighbor queen (x, y) on an adjacent row or column, in minimal distance. We take the translation $H^{-i} \circ V^{-p(i)}$ as first part; that maps the first queen to $(0, 0)$. The neighbor queen (x, y) is then mapped to one of the eight cells $(1, k)$, $(1, -k)$, $(-1, k)$, $(-1, -k)$, $(k, 1)$, $(k, -1)$, $(-k, 1)$, or $(-k, -1)$. If it is at $(1, k)$, we need no further mapping. Otherwise we compose the translation with a vertical reflection and perhaps a rotation, depending on the position of the neighbor queen; the rotation may be 90°, 180°, or 270°.

The second condition is shown by contradiction: if there is a mapping α such that $\alpha(p)(0) = 0$ and $\alpha(p)(1) = k'$, for some $k' < k$, then the minimal neighbor distance of $\alpha(p)$ is less or equal to k'. By virtue of the first lemma, the minimal neighbor distance of $\alpha(p)$ is equal to the minimal neighbor distance of p which is k. Thus $k \leqslant k' < k$, a contradiction.

The sufficiency of the two conditions is clear from the same arguments. □

It is clear that $d_p \leqslant n/2$. If $d_p = 1$, then two ‘queens’ of p are diagonal neighbors, perhaps at the horizontal or vertical margin. That cannot happen for torus solutions, but is possible for regular n-queens solutions.

If $d_{p,i} = 2$ for some $p \in \mathbf{T}_n$ and some i, then the queen at $(i, p(i))$ has a neighbor in the distance of a knight’s move. We generalize that to calculate the second property for the classification, the length of a knight chain in p.

Definition 2.4. A permutation $p \in S_n$ *contains a knight’s chain of length* l, if there is an $\alpha \in \mathbf{D}_{4,n}$ such that $\alpha(p)(k) = kd_p$ for all $k < l$. The *knight length of* p is the maximal l such that p contains a knight’s chain of length l.

By definition of d_p, the knight’s length of a permutation is greater than or equal to 2.

Again, the knight length of p is a class function on $\mathbf{T}_n$.

What happens at the end of the longest chain? That leads to the third property, the ‘secondary minimal distance’. But here we have the problem that the chain may be endless, forming a sort of circle. In this case, the chain has length n and contains all queens, as the first coordinate in the definition has step size 1.

Definition 2.5. Let $p \in S_n$ be a permutation, l be its knight length, and $l < n$. p *has secondary minimal distance* e if there is an $\alpha \in \mathbf{D}_{4,n}$ such that $\alpha(p)(k) = kd_p$ for all $k < l$, and $\alpha(p)(l) = \alpha(p)(l-1) + e \bmod n$, and if e is minimal with that property. If $l = n$, then the secondary minimal distance of p is 0.

Also the secondary minimal distance is a class function.

For some permutations, we use also a fourth and last aspect for the classification. For that, we divide the set of permutations of knight length 2 in two distinct subsets. For a p in the first subset, a queen having one 'knight neighbor' may have a second knight neighbor (in perpendicular direction), and there must be at least one such queen. All other permutations of knight length 2 are in the other subset.

2.5. Splitting search runs and compiling results

Many programs scan the tree of possible solutions, just counting the solutions found. Of course, as the number of solutions increases rapidly, it costs too much space to save them all.

The improved algorithm uses symmetries to store the solutions; in the end, it provides a transversal of all solutions, with respect to some group acting on them.

That allows splitting of the search runs and compiling the results afterwards.

In the search for $\mathbf{T}_{29}$, the author started different runs of the program; the runs used different parameters which were given manually (in parameter files). Mostly, the runs searched for different classes of solutions, in the classification of the previous subsection.

To do that, we need further programs, to sort and merge the raw files generated by the different search runs. These programs transform the solutions to a canonical form, and eliminate solutions found twice or more often.

To control the different runs of the search program, the author's program has implemented several commands. As we refer to them in the next section, they are introduced here. Commands for placing queens and blocking cells statically are described here, further commands in the next subsection.

2.6. Static assignment and blocking of cells

The command `Set 1 1` instructs the program to set a queen on cell 1/1 for the complete run. Several `Set` commands are allowed but they should not contradict each other.

The command `Block 1 1` blocks cell 1/1 for the complete run. No queen may be set on that cell. Several commands of this type are also possible for a run.

2.7. Dynamic blocking of cells

To avoid solutions of orbits that were already found, the algorithm uses dynamic blocking of cells. To be more specific: with every queen set during the run, the algorithm checks whether there is some condition to block further cells. If so, the affected cells will be blocked. They will be unblocked again if the queen raising the condition is removed during backtracking.

The effort for every step increases, but overall the run accelerates significantly as it searches only fewer, more special solutions. It concentrates on finding solutions of orbits that are not yet 'touched'.

Note that the algorithm does not strictly exclude finding solutions of the same orbit twice or several times; but it reduces the number of hits per orbit from say 100 to some value between 2 and 10.

The command `MinDist 4` instructs the program to maintain a minimal distance between the queens on neighbored lines. That is visualized in Fig. 5. Line means row and column here, not including — as above — (anti-)diagonals.

If, for instance, a queen is on 7/9, then with `MinDist 4` no queens may be on the cells marked with 'b', i.e. 6/6, 6/7, 6/11, 6/12, 8/6, 8/7, 8/11, 8/12, 4/8, 5/8, 9/8, 10/8, 4/10, 5/10, 9/10 and 10/10. This command is allowed only once per run.

The command `SecMinDist 6` causes the program to ensure another minimal distance for all pairs of queens that are set in the primary minimal distance. As the previous command, this also makes sense only once per run.

An example is given in Fig. 6. We have the two commands `MinDist 2` and `SecMinDist 6` together with queens on 7/9 and 8/7.

That makes the program block the cells 6/11, 6/12, 6/13, 6/14 on the left side of the 'MinDist pair', and 9/5, 9/4, 9/3, and 9/2 on the right side.

The command `NoKnightLine` tells the program never to set three queens in a 'knight's line'.

For example (Fig. 7), queens on 4/3, 6/4 and 8/5 are forbidden. If there are queens on 4/3 and 6/4, then cell 8/5 is

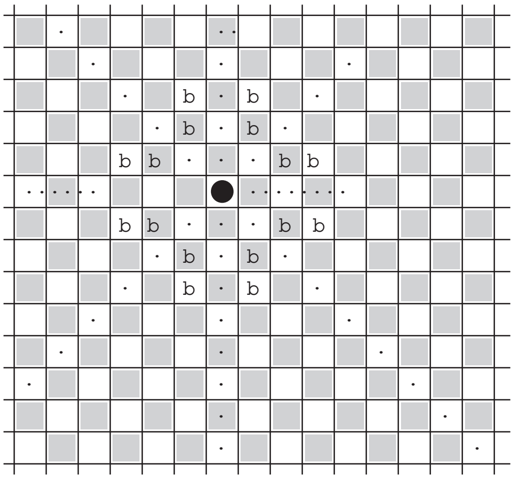

Fig. 5. Cells blocked as effect of command MinDist 4.

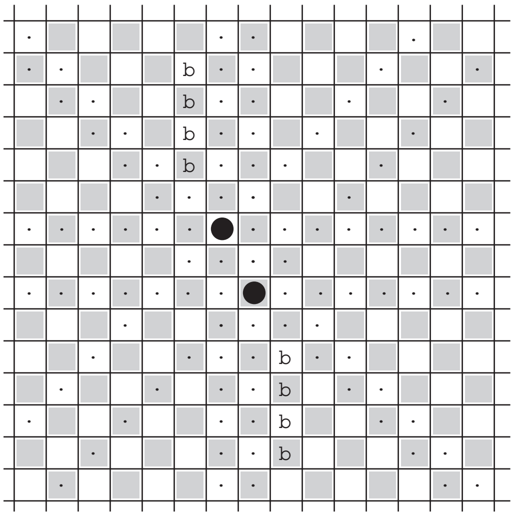

Fig. 6. Cells blocked as effect of command SecMinDist 6.

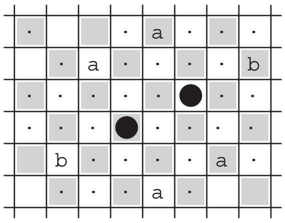

Fig. 7. The two cells that would establish a knight's line, together with the two queens, marked with 'b', are blocked as an effect of the command NoKnightLine. The cells marked with 'a' are still allowed.

blocked, as is cell 2/2. A third queen in perpendicular direction—that would be on 5/1 or 3/5 for the first queen of the example—is allowed.

The command `SingleKnight` forbids, for two queens in knight's distance, also further 'knight neighbors' in perpendicular direction. That is shown in Fig. 8.

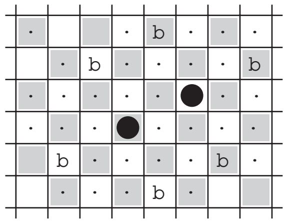

Fig. 8. Effect of command SingleKnight: each queen may have at most one 'knight neighbor'.

2.8. *Further, larger groups of symmetries*

Further groups of mappings which map the chessboard onto itself are of interest. First, we can go from congruent mappings to similar mappings. The author has thought of exploiting the action of the group $\mathbf{Sim}_n$ introduced in 1.3.2 *for the search* but had not yet pursued that idea when this article was prepared. For $\mathbf{T}_{29}$, the action was used only to reduce the space for the transversals which are stored. Having a transversal of torus congruence, we can construct a transversal for similarity.

The use of affine motions on the torus, i.e. of $\mathbf{Aff}_n$ may lead to even more effective search algorithms, but that is still open. At least, it will save space for storing transversals of the solutions.

The problem with $\mathbf{Aff}_n$ is that it does not act on $\overline{\mathbf{T}_n}$. There is an action on the larger set $\mathfrak{P}^n(\mathbf{F}_n)$; if we have an orbit, we must investigate which elements of it are solutions.

By the way: $\mathbf{T}_5$, $\mathbf{T}_7$ and $\mathbf{T}_{11}$ have *only one orbit* under the action of this group. All solutions in these sets are affine images of the main diagonal.

3. Search of $\mathbf{T}_{29}$

3.1. *Remark on the starting point*

The author used the group of congruent motions on the torus. For canonization (i.e. to see simply if a solution was found earlier), the sort and merge program transformed every solution to the lowest element of its orbit, in alphabetic order.

The different steps of the search were oriented on the classification given in 2.4.

3.2. *Steps of the search*

Searching the torus solutions for the 29 × 29 board was done in the following steps:

First, the author searched all solutions that have at least three queens in a 'knight line' of minimal neighbor distance 2. That was done by the following commands for the search program:

```
Set 1 1
Set 2 3
Set 3 5
```

To keep the single runs small, he added a fourth queen in the fourth row. On the other hand, he blocked some cells to make sure that the knight line starts at 1/1, and to avoid solutions found already by their symmetric image. That gives, for instance, the complete command set:

```
Set 1 1
Set 2 3
Set 3 5
Set 4 10
Block 29 28
Block 29 27
Block 29 26
```

Table 1
Number of orbits under action of $D_{4,n}$, depending on size

Size	Number	Size	Number	Size	Number
1	0	29	0	841	0
2	0	58	1	1682	1215
4	0	116	6	3364	121,487
8	0	232	0	6728	89,997,968

In this way, the search avoided—by static blocking of cells—finding too often solutions p with $d_p = 2$, $l_p = 3$ and secondary minimal distance 3 or 4.

Then the author searched solutions having $d_p = 2$ and $l_p = 2$. They have one or several pairs of queens in knight's distance but not in a knight line. The author further divided these cases into cases where one of the queens has a second 'knight neighbor' at right angles to the first, and those cases where it has not. The first case, he sorted as 'NoKnightLine', the second as 'SingleKnight'. In the latter case, which needs most effort, he additionally used `SecMinDist`.

An example for the commands for SingleKnight:

```
Set 1 1
Set 2 3
Set 3 7
SecMinDist 4
SingleKnight
```

and perhaps another fixed queen, just for partitioning the runs.

An example for the commands for NoKnightLine:

```
Set 1 1
Set 2 3
Set 4 2
NoKnightLine
```

and also perhaps another fixed queen for partitioning the runs. These runs used (by the commands) dynamic blocking of cells.

Finally, the author searched the solutions having a higher minimal distance ($d_p > 2$), as 'MinDist'. These runs had the commands, for example:

```
Set 1 1
Set 2 5
Block 29 26
MinDist 4
```

and perhaps more partitioning.

3.3. *Results for* $\mathbf{T}_{29}$

Table 1 gives the number of orbits for the action of congruent motions, depending on the size of the orbit. As each orbit is isomorphic to the quotient group of the original group by the stabilizer of the orbit, all sizes are divisors of the order which is $6728 = 8 * 29 * 29$. The table is organized by the divisors.

Table 2 shows the distribution of the orbits, depending on the search types which we discussed in the previous sections.

From the results in Table 2, we compute in Table 3 the numbers of solutions for each search type and the total number of solutions, 605,917,055,356.

In Table 4, we show the number of solutions, depending on size of orbit or generator of stabilizer; the generator of stabilizer is taken up to a conjugation.

The size of an orbit allows one to say what the generators of the stabilizer are, up to some conjugation; that is because no n-queens solution has M (a reflection) as stabilizer, and also combinations of shift and reflection are impossible for torus solutions, as n must be odd. That is not true for regular n-queens solutions, where n can be even. Indeed, some of

Table 2
Number of $\mathbf{T}_{29}$ orbits depending on the search type

Size	1–3–5	NoKnightLine	SingleKnight	MinDist	Sum orbits
29					0
58				1	1
116	1			5	6
232					0
841					0
1682	128	80	560	447	1215
3364	24,819	9458	74,408	12,802	121,487
6728	13,451,061	17,072,964	58,416,976	1,056,967	89,997,968
Orbits	13,476,009	17,082,502	58,491,944	1,070,222	90,120,677

Table 3
Number of $\mathbf{T}_{29}$ solutions depending on the search type

1–3–5	90,582,444,936	14.9%	NoKnightLine	114,898,853,064	19.0%
SingleKnight	393,280,664,960	64.9%	MinDist > 2	7,155,092,396	1.2%

Table 4
Number of $\mathbf{T}_{29}$ solutions depending on symmetry type

Size	Generator of stabilizer	Number of solutions
58	$H^k \circ V, k \in \{12, 17\}, R$	58
116	$H^k \circ V, 2 \leqslant k \leqslant 27, k \notin \{12, 17\}, R^2$	696
1682	R	2,043,630
3364	R^2	408,682,268
6728		605,506,328,704

the classical 8-queens solutions have $M \circ V^4$ as stabilizer. Therefore, it is natural that we have no orbits of size 29 or 841: they would have some reflection in their stabilizers, and that is impossible for n-queens solutions.

There is also a simple reason why there is no orbit of size 232: such an orbit has some translation in it; for n-queens solutions, or every set of 29 cells, that means that all these cells lie 'in a straight line' and that also has a 180° rotation in its stabilizer.

3.4. Some figures concerning performance

Real performance data would require a discussion of the programming language, of the computers used, and of program versions. That is not given here, especially since the program was refined several times during the $\mathbf{T}_{29}$ search.

Instead, the author did two things: firstly, he did a search for $\mathbf{T}_{17}$ with the final version of the program. That is discussed in the next lines. Secondly, he implemented the standard algorithm in the same environment that was used for the search, and made a comparison of runtimes for T_{23} which is given at the end of this subsection.

The new program always counted the number of backtracks. These are given in the following tables. Table 5 shows the gain due to programming improvements, i.e. look ahead, usage of columns (not only rows) and of diagonals (Sections 2.2 and 2.3).

Table 6 shows the gain achieved by exploitation of symmetry.

For that, the author added five further variants to the first variant. The first variant is only the most elementary symmetry, placing a first queen on cell 1/1 and multiplying the resulting number by 17, given also in the last row of Table 5. The other five variants differ in the number of single runs. For instance, variant 2 uses two runs only: one with fixed queens on 1/1 and 2/3, searching all solutions of `MinDist` $= 2$, and a second with only the queen on 1/1 fixed,

Table 5
Number of backtracks, depending on search algorithm, without use of symmetry

	Full search	First queen at 1/1	Ratio	Gain (%)	Gain ratio	Gain for use col./diagonal (%)
No look ahead	226,060,611	13,297,682	17.0			
No columns	11,359,621	668,212	17.0	94.97	19.9	
No diagonals	9,157,815	538,516	17.0	95.95	24.7	19.41
Without restrictions	7,388,878	434,802	17.0	96.73	30.6	34.93

Table 6
Number of backtracks, depending on different use of symmetry

	Variant 2	Variant 3	Variant 4	Variant 5	Variant 6
1Q + min1Dist 3	44,511				
2Q + min1Dist 3		3873	3873	3873	3873
2Q + min1Dist 4		463	463	463	463
2Q + min1Dist 5		69	69	69	69
1Q + min1Dist over 5		49	49	49	49
2Q (1/1,2/3)	29,757	29,757			
Queen chain 8			11	11	11
Q chain 3 + block 17/16			2137	2137	2137
NoKnightLine			1742	1742	1742
SingleKnight			19,768		
SecMinDist 3				1924	1924
SecMinDist 4				1713	1713
SecMinDist 5				1623	1623
SecMinDist 6				959	959
SecMinDist 7				868	868
SecMinDist over 7				4500	
SecMinDist 8					916
SecMinDist 9					662
SecMinDist 10					618
SecMinDist over 10					324
Total number of backtracks	74,268	34,211	28,112	19,931	17,951
Improvement ratio					
To elementary symmetry	5.9	12.7	15.5	21.8	24.2
To usual search	179	389	473	667	741

and with `MinDist` $\geqslant 3$. Variant 3 splits the run with `MinDist` $\geqslant 3$, and the variants 4–6 split the run for `MinDist` $= 2$ (queens at 1/1 and 2/3).

The run with a queen (knight's) chain of length 8 is inserted only to produce the solution of the endless queen chain. That allows blocking the cell before the queen chain of length 3, in the next line of the table, and avoids finding shifted images of solutions with a longer queens chain too often.

The last line gives the factor that is achieved by both programming improvements and use of symmetry, compared to the basic algorithm. Other numbers not given here show that the factor increases with the size of the board.

As mentioned above, the author did also a comparison of runtime for T_{23} (Table 7). That was done on a 900 MHz Intel computer under Linux (Suse distribution, Version 9.2). The programs are written in Java, JDK 1.5, and ran under Java(TM) 2 Runtime Environment, Standard Edition, (build 1.5.0 06-b05) and JVM Java HotSpot(TM) Client VM (build 1.5.0 06-b05, mixed mode, sharing).

The runs for the T_{29} search were done using the spare time of a 300 MHz Intel computer, over a time of eight months. A coarse guess for the CPU time on that machine is 5000 h.

Table 7
CPU times for T_{23}, depending on algorithm and use of symmetry

	Classical algorithm	New program, only computational improvements	New program using symmetries
CPU time for search (h)	91.3	1.5	0.1
CPU time for search (s)	328,916	5438	372
CPU time for sort and evaluation (s)			29
Number of backtracks	187,806,563,266	1,153,202,933	58,719,054
Backtracks per second	571,000	212,000	157,000

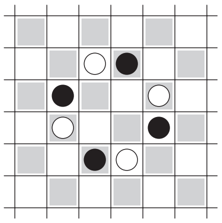
Fig. 9. The white queens occupy the same line set as do the black queens.

4. Algorithm of Rivin and Zabih

4.1. Idea of the Rivin–Zabih Algorithm

Rivin and Zabih [3] give an algorithm that is not a backtracking algorithm.

The idea of this algorithm is to shift the focus from the single queens to the line sets that are occupied in a special situation. There are situations where we come to the same line sets in different ways. A simple example is illustrated in Fig. 9: both the white and the black queens occupy the same set of rows, columns, diagonals and anti-diagonals. There is no influence on further queens; if we have a solution with the white queens, we have a solution with the black queens as well.

The Rivin–Zabih algorithm maintains lists of such sets of lines. Starting with the empty set, it first 'adds' queens in the first row. That means, for every queen that can be placed in the first row, it adds its row, column, diagonal and anti-diagonal to the set of lines, and stores that new line set in the list of line sets.

Then, it tries to combine every line set with queens on the second row, and looks if that results in a line set already found. If so, it joins the possibilities leading to the line set, by maintaining a counter with each line set.

That is continued for every line. If there are solutions, the algorithm finishes with one line set containing all rows, columns and (anti-)diagonals, in the torus case. The counter associated with that line set gives the number of solutions. In the regular case, the final line sets may differ in their diagonals or anti-diagonals. The number of solutions is the sum of their counters.

The author has implemented that algorithm. However, it was not of practical use until now; it requires too much space. A search for $\mathbf{T}_{17}$ failed with 500 MB of main memory; it was possible only after the line sets were written to disk in portions, and joined there after a row was finished.

Can this algorithm be improved, as it was done with the backtracking algorithm? Two developments can be made easily.

4.2. Further look ahead

A better look ahead can reduce the number of possible line sets in the lists. Given a line set and a free line, we have to find out if there are still free cells on that line (recall the slightly different meaning of free for lines and cells, introduced at the beginning of Section 2.1).

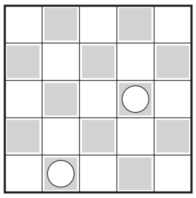

Fig. 10. A simple conflict; the two queens attack each other on the diagonal.

To be more precise: if L is the set of all rows, columns, diagonals and anti-diagonals, $l \in L$ and if a line set G is given, then

$$f_{l,G} := l \Big\backslash \bigcup_{g \in G} g \tag{12}$$

is the set of free cells of line l for G. If $f_{l,G}$ is empty for some $l \notin G$, then G can be deleted from the list of possible line sets.

That idea can be extended: if $|f_{l,G}| = 1$ for some $l \in L \backslash G$, then we can place queens on all such cells; we get a new line set G' from that. The test for exhausted lines may be iterated with that G', if $G' \neq L$, giving perhaps a G'', as long as $|f_{l,G}| = 1$ for some $l \in L \backslash G'$. The line sets $G', G'' \ldots$ are constructed just for lookahead; they are not entered in the list of possible line sets. The original G either remains unchanged in the list, or is deleted from it if some G' indicates so.

Another idea is to be more flexible in the order of lines; however, that is probably restricted to the order of rows. Proceeding also with columns and diagonals or anti-diagonals is complicated, though possible. It has the disadvantage that we have line sets stemming from different numbers of queens in the list. That is simpler in the backtrack algorithm, as the data are more 'local' in some sense.

One thing can be done easily: each line set G left after the above deletion can give a 'vote' for the free rows, just by $|f_{l,G}|$, for the free row l, or by some weight function of $|f_{l,G}|$. The next row is chosen depending on the sum of these values for all remaining G.

4.3. Splitting runs and compiling results

The idea is the same as in the backtracking algorithm. We start with a given line set, stemming from the fixed queens. However, the algorithm must be extended: in its original version, it only counts solutions; now we need the set of solutions. For that, some information must be stored every time that two line sets are joined, i.e. where we found different ways to come to the same line set. A further program should then generate the solutions from that information.

It is an open point whether the Rivin–Zabih algorithm is of practical use with these enhancements.

5. Application to the regular *n*-queens problem

A basic idea for using group actions also for the regular case is to view a regular solution as a torus solution with some flaws; we call these flaws 'conflicts'.

A conflict means that two queens are placed on the same torus (anti-)diagonal. These conflicts can be resolved, i.e. the conflicting queens separated, when the solution is shifted so that the two queens are placed on different regular diagonals, although they are still on the same torus diagonal. We illustrate that with Figs. 10 and 11. In Fig. 11, the rightmost column is repeated on the left side, outside of the regular chessboard.

By moving the queens two cells to the left, the conflict is *separated*; the two queens still attack each other as toroidal queens, but no longer as regular queens. The attack line, i.e. the diagonal that the two queens share, is broken by the margin.

There may be more than one conflict in a regular n-queens solution. In fact, the old argument of Polya, also given in the article of Rivin et al. [2], shows that there must be at least one conflict on the diagonals and a second on the anti-diagonals if n is even.

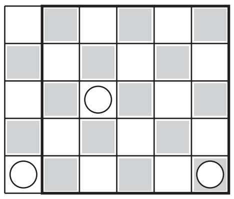
Fig. 11. The same conflict separated.

That leads to sets of conflicts, or conflict constellations. As conflict constellations are basically subsets of $\mathbb{Z}_n \times \mathbb{Z}_n$, we have natural action of our groups also on these conflict constellations. That leads to the following idea for the algorithm: we can first enumerate the conflict constellations; in the second step, we eliminate all conflict constellations that cannot be resolved or separated in any element of its orbit; and in a third step, we complete the separable conflict constellations to complete solutions.

Another interesting point in this context is conflict constellations of three cells. The three cells come from at least two pairs, so one cell must be part of two conflicts. It is impossible for the two other elements also to form a conflict. So, if we take out the queen belonging to both conflicts, we get a valid placing of $n - 1$ torus queens. That is what Schlude and Speaker investigate in [5]. They show that such placements are not always possible, and give conditions when they exist.

The handling of conflict constellations might also be a feasible way to handle the regular n-queens problem.

6. Final remark

We have shown that the use of finite group actions can improve the search algorithms for both $\mathbf{T}_n$ and $\mathbf{Q}_n$. The case of regular n-queens solutions $\mathbf{Q}_n$ needs some extra effort. The new aspects may also be combined partly with the Rivin–Zabih algorithm.

Although n-queens solutions are quite simple combinatorial structures, there are still interesting open points.

Acknowledgment

I wish to thank Eric Sprigg, a former colleague, who helped to bring this article in a readable English form.

References

[1] A. Kerber, Applied Finite Group Actions, second edition, Springer, Berlin, 1999 ISBN 3-540-65941-2.
[2] I. Rivin, I. Vardi, P. Zimmermann, The n-queens problem, Amer. Math. Monthly 101 (1994) 629–639.
[3] I. Rivin, R. Zabih, A dynamic programming solution to the n-queens problem, Inform. Process. Lett. 41 (1992) 253–256.
[4] U. Schimke, E-Mail communication.
[5] K. Schlude, E. Specker, Zum Problem der Damen auf dem Torus, Technical Report 412, Computer Science, ETH Zürich, 2003 (in German).