Discrete Applied Mathematics 158 (2010) 308–318

Contents lists available at ScienceDirect

Discrete Applied Mathematics

journal homepage: www.elsevier.com/locate/dam

MINIMUM DOMINATING SET OF QUEENS: A trivial programming exercise?

Henning Fernau

Universität Trier, FB IV—Abteilung Informatik, 54286 Trier, Germany

ARTICLE INFO

Article history:
Received 9 November 2007
Received in revised form 4 March 2009
Accepted 30 September 2009
Available online 20 October 2009

Keywords:
Exact algorithms
Parameterized algorithms
$\mathcal{NP}$-completeness
Chess problems
Domination problems

ABSTRACT

MINIMUM DOMINATING SET OF QUEENS is one of the typical programming exercises of a first year's computer science course. However, little work has been published on the complexity of this problem. We analyse here several algorithms and show that advanced algorithmic techniques may dramatically speed up solving this problem.

1. Introduction

Problem definition. MINIMUM DOMINATING SET OF QUEENS is the minimization version of the following decision problem:

Problem name: DOMINATING SET OF QUEENS (QDS)
Given: An $n \times n$ chessboard C
Parameter: a positive integer k
Output: Is it possible to place k queens on C such that all squares are dominated?

Our Motivation. MINIMUM DOMINATING SET OF QUEENS is one of the typical programming exercises of a first year's computer science course. However, little work has been published on the complexity of this problem.

We will discuss three approaches in the following sections:

- backtracking,
- dynamic programming on subsets,
- dynamic programming based on path decomposition.

This shows how various algorithmic techniques could be presented and explained through this one example problem, which is also both illustrative, motivating and challenging to students. While the first technique is the one expected from first year's computer science students, the other two techniques are (much) more advanced. In particular, the last technique is thought to be pretty hard to explain to and to grasp for typical computer science students. We show that this need not be the case with this particular example, since there are close relations with the possibly better known sweep line techniques known from algorithmic geometry.

At the end of the paper, we briefly discuss some complexity issues regarding this problem.

E-mail address: fernau@uni-trier.de.

0166-218X/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.dam.2009.09.019

Combinatorial Background. One instance is the so-called *Five Queens Problem* on the chessboard[1]: it is required to place five queens on the board such that they dominate each square. This task corresponds to DOMINATING SET as follows: the squares are the vertices of a graph; there is an edge between two such vertices x, y iff a queen placed on one square that corresponds to x can directly move to y (assuming no other pieces on the board). Hence, the edge relation models the way a queen may move on a chess board, yielding the *queen chessboard graph*.

According to [9], the general version played on a (general) $n \times n$ board is also known as the *Queen Domination Problem*, or *Board Covering Problem* [26]. The task is to find the minimum number of queens that can be placed on a general chessboard so that each square contains a queen or is attacked by one. This amounts to finding the minimum domination number $\gamma(Q_n)$ of the general *queen chessboard graph* Q_n. Recent bounds on the corresponding domination number can be found in [7,8,10,25]. It appears that the queen domination number is "approximately" $n/2$, but only few exact values have been found up to today. In fact, the exact characterisation of this number sequence $\gamma(Q_n)$ is listed as Problem C18 within Guy's book on "Unsolved Problems in Number Theory", see [19]. It is known [16,25] that $\gamma(Q_n) \geq \lceil n/2 \rceil$ for all positive integers n except for $n = 3, 11$. Regarding upper bounds, the best asymptotic bound appears to be $\gamma(Q_n) \leq 69n/133 + \mathcal{O}(1)$. However, the $\mathcal{O}(1)$-constant involved makes it hard to use this upper bound for determining concrete values of $\gamma(Q_n)$. Notice that the mentioned asymptotic upper bounds are essentially derived by "small" concrete patterns for boards Q_ℓ that have to be repeated somehow in order to obtain domination patterns for $Q_{c\ell}$. Solutions for graphs Q_n with n not being a multiple of ℓ are derived from the largest $c\ell < n$, by filling up the "remaining" $n - c\ell$ rows with queens. So, the $\mathcal{O}(1)$-term could be as large as 100, possibly growing further when new (and larger) patterns are discovered in order to further improve on the asymptotic estimate for $\gamma(Q_n)$.

Problem Variants. By way of contrast, observe that on the $n \times n$ square *beehive*, the queen domination number has been established to be $\lfloor (2n+1)/3 \rfloor$, see [30]. It is clear that similar domination-type problems may be created from other chess pieces as rooks, bishops, or knights (or even artificial chess pieces), see [17] as an example.

Also the variant that only certain (predetermined) squares need to be dominated has been considered. For example, the $n \times n$ *Rook Domination Problem* is trivially solvable by choosing one main diagonal and putting n rooks on this diagonal. By pigeon-hole, any trial to dominate all squares by fewer than n rooks must fail. However, if only a certain number of predetermined squares need to be dominated by placing a minimum number of rooks on one of those predetermined squares, we arrive at a problem also known as MATRIX DOMINATION ; see [15] for further results on this problem. Further variants of the Queen domination problem are discussed in [10].

A related problem on the chessboard is the following one, originally introduced in 1850 by Carl Friedrich Gauß, see pp. 19–21 in [18]: the *n-Queens Problem*. This problem may be stated as follows: find a placement of n queens on an $n \times n$ chessboard, such that no queen can be taken by any other, see [27]. Again, this problem can be also considered for other chess pieces and modified boards, see (e.g.) [3] as a recent reference. If (for the moment) we neglect the well-known property that in fact a solution exists to the n-Queens Problem for each n (but $n = 2, 3$), then the n-Queens Problem corresponds to the task of solving MAXIMUM INDEPENDENT SET on the queen chessboard graph. In fact, this approach was taken in [24], where an integer linear programming formulation was presented. However, there are quite simple algorithms that yield solutions for that problem for any n, as explained by the Wikipedia entry: http://en.wikipedia.org/wiki/Eight_queens_problem. A nice treatment of both the independence and the domination problem can be also found in http://mathworld.wolfram.com/QueensProblem.html.

2. A solution by backtracking

First, we are going to describe a solution that is the one students are most likely to come up with when asked to write a program for MINIMUM DOMINATING SET OF QUEENS, in particular when this is done in the context of teaching backtracking, see Alg. 1.[2]

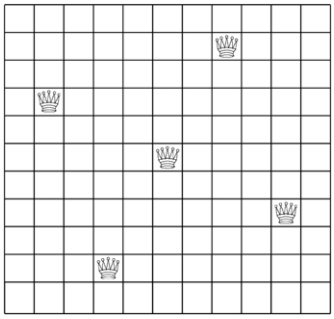
The fact that algorithmics in exponential time matters possibly more than in polynomial time can be underlined by the fact that, while the recursive backtracking program I wrote for solving MINIMUM DOMINATING SET OF QUEENS took about ten minutes for optimally solving an $n \times n$-board with $n = 10$, it took more than 4.5 h with $n = 11$ (measured on a rather old ThinkPad notebook). Then, I ran out of patience. Fig. 1 shows the solution the program found:

The chessboard might be represented as a two-dimensional array b with the following meaning (this also explains the right-hand table above):

- If $b[i, j] = 0$, then no queen is on this square, nor is any queen attacking the square.
- If $b[i, j] = k > 0$, then queen no. k is residing on that square.
- If $b[i, j] = -k < 0$, then queen no. k is attacking that square, and no other queen with a number smaller than k is attacking that square; moreover, no queen is residing on that particular square.

[1] The terminology in this area is a bit awkward: as can be seen, the Five Queens Problem is not a special case of the n-Queens problem introduced below. We will rather distinguish these problems by referring to them as the Queen domination problem and the Queen independence problem later on.

[2] We ignore the additional possibility of testing all 0-1 assignments to all squares, since the corresponding naive $\mathcal{O}(2^{n^2})$ algorithm is no contender for our presentation.



-3	-1	-5	-2	-1	-3	-4	-4	-4	-5	-3
-4	-1	-4	-1	-4	-3	-4	4	-4	-3	-4
-1	-1	-1	-2	-5	-3	-4	-4	-3	-5	-2
-1	1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-2	-3	-3	-3	-4	-2	-5	-4
-3	-1	-3	-1	-3	3	-3	-2	-3	-3	-3
-2	-1	-4	-2	-1	-3	-2	-4	-5	-5	-5
-5	-1	-5	-2	-5	-1	-5	-3	-5	5	-5
-4	-1	-2	-2	-2	-3	-1	-4	-3	-5	-5
-2	-1	-2	2	-2	-2	-2	-1	-2	-2	-2
-3	-1	-2	-2	-2	-3	-5	-4	-1	-5	-3

Fig. 1. One of the solutions to MINIMUM DOMINATING SET OF QUEENS on a 11×11 board.

Algorithm 1 A backtracking algorithm for MINIMUM DOMINATING SET OF QUEENS, called QDBT

Input(s): an integer pair $(i, j) \in \{1, \ldots, n\} \times \{1, \ldots, n\}$; variable q stores the number of queens used so far in the current recursion branch; no-zeros records the number of non-dominated squares on board b in the current situation; qrecord is a global variable that stores the record solution found up to this point; b is a global array that stores the current queen placement.

Output(s): an optimum solution of MINIMUM DOMINATING SET OF QUEENS on an $n \times n$ board

```
if q ≥ qrecord then
  return; {no improvements possible}
end if
{1st case: queen on square (i, j)}
put-queen(i, j); update no-zeros;
if no-zeros == 0 AND q < qrecord then
  update qrecord;
  return; {any other solution on this branch cannot be better}
end if
compute next board coordinates (i', j');
return if (i', j') are not within the board;
call QDBT(i', j', q + 1, no-zeros);
{2nd case: (i, j) is empty}
remove-queen(i, j); update no-zeros; call QDBT(i', j', q, no-zeros);
```

What is the running time of Algorithm QDBT? At first glance, it looks as if basically all possibilities of putting queens on the chessboard are investigated, which gives the quick upper bound of $\mathcal{O}(2^{n^2})$. But is it really the case that the actual running time is that bad? Observe that we will recursively fill up one line of the chessboard with queens in a depth-first fashion (actually leaving one square empty), which immediately bounds the number qrecord by $n - 1$. Since we are constantly checking if the number q of queens is smaller than qrecord, we will never place more than $n - 1$ queens on the board.

Theorem 1. *Alg.* 1, *when initially called with* $(i, j) = (1, 1)$, $q = 0$, *qrecord* $= n$, *and no-zeros* $= n \times n$, *runs in time* $\mathcal{O}(c^{n \ln(n)})$ *for some constant* c.

Proof. The correctness of the description of the algorithm should be clear with the comments and annotations given. In particular, since putting a queen on each square of the first row would create a trivial valid (and known) solution, we can safely instantiate qrecord by n, so cutting off all valid solutions that might be found by putting more queens on the board.

As to the running time, a simple estimate is that at most all at most n-element subsets of an n^2 element universe are tested. This leads to a run time estimate of $\mathcal{O}(n^{2n}) = \mathcal{O}(c^{n \log n})$ for some c. ■

Remark. We could improve the estimate given in the proof of the preceding theorem slightly due to Stirling's formula:

$$\binom{n^2}{n} = \frac{n^2(n^2-1)\cdots(n^2-(n+1))}{n!} \leq \frac{n^{2n}}{n!} \approx \frac{(en^2)^n}{n^n} = (en)^n = \mathrm{e}^{n(1+\ln(n))}.$$

Can we do better? In practice, for sure. The techniques described by Östergård and Weakley [25] obviously provide tremendous speed-ups. However, we are not aware of any proof that shows that, speaking in asymptotic terms, their algorithm runs faster than $\mathcal{O}(c^{n \log n})$ for some constant c. Their technique is basically relying on a cute pointer management in backtracking algorithms, also see [23].

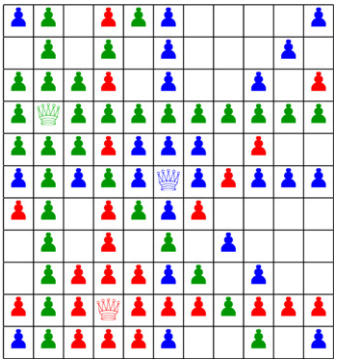


Fig. 2. An illustration of the notion of domination pattern. Let us illustrate how sets of lines (horizontal, vertical, diagonal) yield *domination patterns*. In the picture to the right, three queens are positioned. The squares they dominate are occupied by pawns thus also indicating which queen is dominating that particular square. The set of all squares that are occupied by pawns or queens in this illustration is the domination pattern corresponding to the 12 lines that are directly dominated by the three queens.

We can get better algorithms by using more advanced algorithm techniques. This also proves that MINIMUM DOMINATING SET OF QUEENS can be not only used to teach backtracking, but may be also employed for exemplifying other algorithmic techniques, as well. This will be detailed in the following two sections.

3. Dynamic programming

Generally speaking, dynamic programming can be seen as a method that gradually combines optimum solutions found for sub-structures to optimum solutions for larger structures, up to the point that an optimum solution for the originally given problem is found. When trying to apply this technique, one has to define what is understood under a "sub-structure" in the particular setting, and how to combine optimum solutions.

In our case, we consider sets of lines, i.e., rows, columns, or diagonals, as describing the sub-structures, namely the set of squares that they contain. We then store the minimum number of queens that are necessary to dominate exactly those lines, seen as set of squares (and no other squares of the board). So, many sets of lines cannot be dominated in this particular, exclusive way.

The main observation for the success of the dynamic programming approach is the following one: for each set of lines (be them horizontal, vertical, or diagonal), the information how many queens are sufficient to dominate those lines is basically sufficient to gradually compute the minimum number of queens required to dominate an $n \times n$ board. Since there are n horizontal, n vertical, and $4n - 2$ diagonal lines, in total subsets of less than $6n$ lines have to be maintained, tentatively leading towards an $\mathcal{O}(c^n)$ algorithm for MINIMUM DOMINATING SET OF QUEENS for some c.

However, since different sets of dominated lines can lead to the same set of dominated squares, we must do our bookkeeping in terms of those dominated squares, called *domination patterns*. Naively, this would mean that we maintain an array indexed by subsets of dominated squares where we store the number of queens that are needed to dominated exactly those squares. However, this naive approach would again lead to resource requirements that are of order 2^{n^2}. Domination patterns are illustrated by means of an example in Fig. 2.

Conversely, consider the mapping g that associates to a set of lines the corresponding domination pattern, i.e., $g(S) = \bigcup_{\ell \in S} \ell$, where S is a set of lines ℓ and a line ℓ is viewed as a set of squares. Since there are at most c^n sets of lines, there are also at most c^n sets of squares in the range of g, neglecting polynomial terms. Then, we run Alg. 2. Finally, we can look up the desired result at $g(M)$, where M is the set of all lines.

In Alg. 2, we maintain a queue that contains information in the form of pairs $(g(S), i)$ where partial solutions are encoded.[3] We will refer to the first component of this pair also as the key index. More specifically, S is a set of already dominated lines, and i shows how many queens are needed to *exactly* dominate the squares in the lines of S. This semantics justifies the initialisation: the empty set is dominated by zero queens. Of course, one could also in addition store information about the partial solution (dominating set) that attains the value i within the data structure pertaining to the key index.

In the outer **for**-loop, we test what happens if we put a queen on a particular square s. This could of course only affect lines that contain s. So, in the inner **for**-loop, we update all sets of squares G, used as key index for pairs (G, i) that can be described by sets of lines. There is one further trick used in Alg. 2: The pairs (G, i) are stored as vertices of a directed acyclic graph called G_Q that represents the Hasse diagram of the inclusion relation given on the key indices, with the empty set representing the root that is displayed at the very bottom of the Hasse diagram. Hence, once having found $(g(S), i)$, we have to look for $(g(S \cup T), j)$ only "above" $(g(S), i)$, using the mentioned digraph structure G_Q.

[3] A similar idea was proposed in [28] for a related queen's problem. However, they did not need to use the function g.

What is the running time of Alg. 2? All operations but the loop through all line subsets take polynomial time. Since for each S, at most one element $(g(S), i)$ will be present in the queue, there are at most 2^{6n} elements ever showing up in the queue at the same time. Clearly, each subset of lines S will cause modifications at most $\mathcal{O}(|S|^4)$ times (choose any subset T of at most four lines from S that, together with $S \setminus T$, might cause an update).

Algorithm 2 A dynamic programming algorithm for MINIMUM DOMINATING SET OF QUEENS

Input(s): positive integer n
Output(s): minimum number of queens necessary to dominate all squares on an $n \times n$ board

```
Set Q to {(∅, 0)}.
{Q should be organised as a directed acyclic graph G_Q reflecting the inclusion relation of the first components (key index);
the "root" of G_Q is always (∅, 0). A directed spanning tree of G_Q is used to investigate G_Q in the second for-loop.}
for all squares s of the chessboard do
  Let T be the set of (at most four) lines that contain s.
  for all line sets S do
    {Start the search through G_Q at the root}
    Look for some (g(S), i) in Q using G_Q.
    if NOT g(T) ⊆ g(S) then
      {putting a queen on s would dominate new squares}
      Look for some (g(S ∪ T), j) in Q using G_Q.
      if found then
        Replace (g(S ∪ T), j) by (g(S ∪ T), min{j, i + 1})
      else
        Add (g(S ∪ T), i + 1) to Q (maintaining G_Q)
      end if
    end if
  end for
end for
Let M be the set of all possible lines.
Find (g(M), j) in Q.
return j
```

We can further improve on this estimate by noting that $\gamma(Q_n) \leq 69n/133 + \mathcal{O}(1)$. Therefore, the number of lines that can be dominated is at most

$$\left[\binom{n}{n/2} \cdot \binom{2n}{69n/133}\right]^2 \approx 39.51^n.$$

The improvement comes from the fact that from the $2n$ $\swarrow$-diagonals, at most $69n/133$ can be covered by putting a queen on it. The same bound is true for the $\nwarrow$-diagonals. However, no improvement shows up for the rows and columns; by elementary combinatorial properties of binomial coefficients, the worst case is assumed when trying to arrange $n/2$ queens on the board. Notice that the bound $69n/133$ is purely combinatorial; if the actual bound could be shown to be $n/2$ (up to some additive constant), then the exponential base would further improve to 37.93 (instead of 39.51), again based on the combinatorial estimate for the diagonals.

Theorem 2. MINIMUM DOMINATING SET OF QUEENS *can be computed in time* $\mathcal{O}(39.51^n p(n))$ *when using Alg.* 2*, where* p *is a polynomial.*

Can we further improve on this algorithm that is based on dynamic programming on subsets? In the case of counting all possible solutions of the Queen independence problem, in [28] arguments were developed that helped reduce the number of lines whose subsets should be considered when introducing a new candidate position from $6n$ down to $3n$. We are not aware of similar tricks in the case of MINIMUM DOMINATING SET OF QUEENS.

However, notice that the bound mentioned in Theorem 2 is not only a bound on time, but basically also a bound on the space consumption. In exponential algorithms, this is the really prohibitive part. We can reduce the space consumption to $\mathcal{O}(19.76^n p(n))$ by fixing in an outer loop the rows on which we will put queens. In the loop that goes through all squares of the chessboard, we can now safely ignore squares that belong to rows that have been decided not to carry queens. If we process the remaining squares "row by row", then each time after considering n squares, a whole row has been processed. By using only one temporary additional bit, we can then see (and check) if actually the row under consideration was filled with a queen or not; if not, the corresponding sets of lines must be deleted. In practice, it would be also quite easy to incorporate some symmetry information within the selection of rows, saving another factor of two.

4. Treewidth technique

4.1. General definitions

A graph of small treewidth, or from an algorithmic point even more interesting, of small pathwidth, is appealing since it allows different but standard forms of dynamic programming techniques to be applied, often generalising algorithms as known on trees or paths. This notion can be formalised as follows.

Definition 1. Let $G = (V, E)$ be a graph. A *tree decomposition* of G is a pair $\langle \{X_i \mid i \in I\}, T\rangle$, where each X_i is a subset of V, called a *bag*, and T is a tree with the elements of I as nodes. The following three properties must hold:

1. $\bigcup_{i\in I} X_i = V$;
2. for every edge $\{u, v\} \in E$, there is an $i \in I$ such that $\{u, v\} \subseteq X_i$;
3. for all $i, j, k \in I$, if j lies on the path between i and k in T, then $X_i \cap X_k \subseteq X_j$.

The *width of the tree decomposition* $\langle \{X_i \mid i \in I\}, T\rangle$ equals

$$\max\{|X_i| \mid i \in I\} - 1.$$

The *treewidth* of G is the minimum k such that G has a tree decomposition of width k, also written $\mathrm{tw}(G)$ for short.

By putting edges into the bags and connecting the bags by the edge-incidence relation of the edges they contain, it is straightforward to see that trees have treewidth one. Further, notice that the second condition implies that each bag forms a separator in the original graph.

If the tree underlying a tree decomposition happens to be a path, one also speaks about a *path decomposition*, and accordingly the notion of *pathwidth* is coined. Many interesting details (and lots of references) on treewidth and pathwidth can be found in Ref. [5,6,22].

Let us first sketch how to algorithmically exploit that we know a path decomposition of small width for a particular graph G. (Tree decompositions are treated alike, but show some additional difficulties.) We work along the path of the path decomposition unidirectional, say from left to right. To each bag, a table is associated in which assumptions on the current situation are made, e.g., describing that a queen is put on some specific vertices of the bag. These situations are used as indices of the mentioned table, in which (by induction) the optimum value of the problem on the graph considered so far is stored (i.e., including all vertices of the current bag and all vertices that are contained in bags "to the left" of the current bag). At the very beginning (and as induction base), the values for the leftmost bag in the path decomposition are computed. Then, dynamic programming is used to compute the values for the next bag B', based on the table associated to the present bag B and considering all possibilities of situations for the vertices from the symmetric difference of B and B'. It is known that one can restrict one's attention to so-called *nice decompositions*, meaning in the case of path decompositions that the mentioned symmetric difference always contains exactly one element. As it will become clearer later, the estimates for the table size (i.e., the memory requirements) and for the running times are the same for the path decomposition methodology, ignoring polynomial terms. By induction, from the values stored in the table associated to the rightmost bag of the path decomposition, the optimum value for the whole graph G can be read off. Such decomposition based algorithms can be easily modified to actually yield a minimum solution (not just its size), by storing intermediate optimum solution sets within the table, as well.

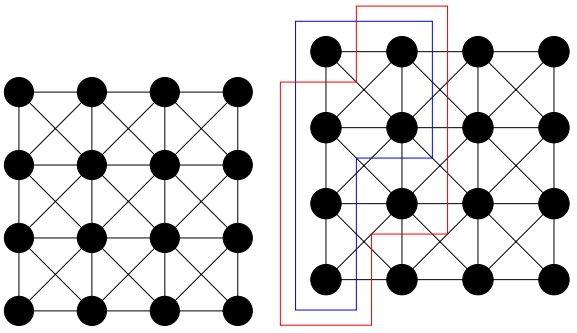
It might be tempting to try to adapt the well-known tree decomposition based algorithms for MINIMUM DOMINATING SET, applying them to Q_n. However, although the structure we started with, namely the chessboard, is of course a planar structure, the queen chessboard graph is *not* planar: in particular, all vertices of Q_n that correspond to one line on the chessboard form a clique of size n, i.e., the family of queen chessboard graphs contains arbitrarily large cliques, quite impossible for planar graphs by Kuratowski's theorem. Only few concrete numbers (or lower bounds) are known for the treewidth of Q_n. In [6], it is reported that $\mathrm{tw}(Q_5) = 14$ and that $\mathrm{tw}(Q_8) \geq 35$. Therefore, we report on a lower bound result of Daniel Lokshtanov (together with his proof), which even holds for the *rook's graph* R_n (corresponding to rook movements), which is obviously a subgraph of Q_n, so that the result easily transfers to Q_n.

Lemma 1. $\forall n : \mathrm{pw}(R_n) \geq n^2/2 - 2$.

Proof. Consider a nice path decomposition of R_n of width smaller than $n^2/4$. Being nice, we can find a bag B with less than $n^2/4$ vertices, such that the graph vertices V_1 that are contained in bags to the left of B (but not in B) are (possibly ± 1) as many as the graph vertices V_2 that are contained in bags to the right of B (but not in B). Notice that B is a separator, hence, removing B from R_n disconnects V_1 and V_2. This implies that, when viewed as squares, the vertices of V_1 cannot lie on the same rows (nor columns) as those of V_2 do. Hence, V_i ($i = 1, 2$) contains squares that are in at most $\lfloor n/2 \rfloor$ different rows (and then, V_{3-i} has only $\lceil n/2 \rceil$ different rows at its disposal). The same is true for the columns, so that altogether

$$|V_1| + |V_2| \leq (\lfloor n/2 \rfloor \rfloor)^2 + (\lceil n/2 \rfloor \rceil)^2 \leq n^2/2 + 1.$$

Conversely, this means that $|B| \geq n^2/2 - 1$. ■



(a) The grid graph G'_4 with diagonals.

(b) Indicating a path decomposition of G'_4.

Fig. 3. Grids with diagonals: graphs with pathwidth $\mathcal{O}(\sqrt{|V|})$.

A similar argument can be used to show that $\mathrm{tw}(R_n) \in \Omega(n^2)$.

Therefore, we would not get $\mathcal{O}(c^n p(n))$-algorithms for any polynomial $p(\cdot)$ and any constant c, when using tree or path decomposition techniques on Q_n directly. However, this is what we could hope for when working with planar graphs $G = (V, E)$ that are known to have a treewidth bounded by $\mathcal{O}(\sqrt{|V|})$, see [12].

4.2. The $n \times n$ planar grid with diagonals

In what follows, we rather consider the $n \times n$ planar grid with diagonals G'_n that corresponds to a chessboard in a straightforward interpretation. In terms of chess piece movements, this graph is rather the King's graph than the Queen's graph.[4] As we will see, the graph G'_n has pathwidth $n+1$. Hence, we will work out an algorithm for computing $\gamma(Q_n)$ that is employing a path decomposition of the King's graph G'_n.

Let us define the *$n \times n$ grid graph with diagonals* $G'_n = (V_n, E'_n)$ as follows:

- $V_n = \{(i, j) \mid 1 \leq i, j \leq n\}$. V_n can be viewed as indices of a square-like grid.
- $E'_n \ni \{(i, j), (i', j')\}$ if one of the following condition is true:
  1. $i' = i + 1$ and $i < n$ and $j' = j$, OR
  2. $j' = j + 1$ and $j < n$ and $i' = i$, OR
  3. $i' = i + 1$ and $i < n$ and $j' = j + 1$ and $j < n$, OR
  4. $i' = i + 1$ and $i < n$ and $j' = j - 1$ and $j > 1$.

  There are no other edges in E'_n than the ones described.

The graph G'_4 is shown in Fig. 3(a). Notice that the first and second condition on possible edges characterise the usual grid graph $G_n = (V_n, E_n)$ (without diagonals).

How does a path decomposition of G'_n look like? Two bags of a possible path decomposition are shown in Fig. 3(b) in the case $n = 4$. Only the first bag contains the left upper corner (with coordinate $(1, 1)$). This vertex $(1, 1)$ has no connection to any vertex outside the bag and can hence be deleted. More generally, define $\sigma(r) = (i, j)$, where $i = (r \bmod n) + 1$ for $1 \leq r \leq n^2$, and $r = (j-1)n + i$. So, $\sigma : \{r \mid 1 \leq r \leq n^2\} \to \{(i, j) \mid 1 \leq i, j \leq n\}$ provides a bijection whose inverse can be seen as a linearization of the grid layout. Define $B(r) = \{\sigma(r + \nu) \mid 0 \leq \nu \leq n + 1\}$, with $1 \leq r < n^2 - n$. It can be seen that the bags $B(r)$ form a path decomposition of G'_n. In particular, $B(r + 1)$ separates $\sigma(r)$ from $\sigma(r')$ for $r' > r + n + 2$, and each edge is represented in some of the bags. The upper bound of the pathwidth is linearly growing with n, i.e., with the square root of the number of vertices of G'_n.

4.3. Dynamic programming on the grid

For our dynamic programming algorithm based on the described path decomposition (solving the problem in time $\mathcal{O}(c^n)$ for a c to be determined), we endow each vertex of this graph (i.e., each square on the chessboard) within the bag with the following information: for each direction (queen movement) of that specific square we have to keep track of how it might still "influence" the yet non-examined squares. Naively, this is done as follows: To each vertex of the bag, we associate a 9-bit-vector $b = (b_{\swarrow}, b_{\leftarrow}, b_{\nwarrow}, b_{\uparrow}, b_{\nearrow}, b_{\rightarrow}, b_{\searrow}, b_{\downarrow}, q)$. Consider now a vertex in the first column of that bag. Then, the first four components describe the possible situations according to the "past" of the computation of the graph, i.e., whether the square (represented by the vertex) is known to be dominated by some queen from the south-west (then, $b_{\swarrow} = 1$), from the west $b_{\leftarrow}$, from the north-west $b_{\nwarrow}$, or from above $b_{\uparrow}$. The next four components describe something about the "future" of

[4] The notation K_n is already reserved in graph theory terminology for the complete graph and can hence not be used for denoting the King's graph.

the computation, e.g., $b_{\rightarrow} = 1$ iff we expect that this square will be dominated from the right. $q = 1$ means that a queen is placed on that particular square. Hence, to a specific bag we can associate a table of size $2^{9(n+2)}$. Each table entry indicates the minimum number of queens that is necessary to dominate the queen graph processed so far, assuming the specific bit vector settings. This means, in particular, that there need not be a queen on any line that has been partially processed so far (say, a particular row r) if we consider the case when the according bit vector(s) associated to the square(s) of the current bag in row r show a one in bit $b_{\rightarrow}$.

We can assume (and check if necessary) *consistency* of the table, marking inconsistent situations with ∞ (represented by a suitable large integer) as a table entry. For example, if there are two squares x and x' in the bag such that x' is the right neighbour of x, then bit vector b of x with $b_{\leftarrow} = 1$ can only be consistent with bit vectors b' of x' where $b'_{\leftarrow} = 1$. Conversely, $b'_{\leftarrow} = 1$ is only possible if $q = 1$ or if $b_{\leftarrow} = 1$. There are also some clearly inconsistent situations from the geometry of the chessboard. For example, a square in the upper row cannot be dominated from any northern direction.

How to update the information can be best described when assuming that we intercalate one more bag $B'(r)$ in the path decomposition between $B(r)$ and $B(r+1)$ by setting $B'(r) = B(r) \cap B(r+1)$. Then, table $t'(r)$ (corresponding to $B'(r)$) is obtained from $t(r)$ (associated to $B(r)$) by computing, for each $(n+1) \times 9$-bit-vector $\vec{b} = (b^1, \ldots, b^{n+1})$,

$$t'(r)(\vec{b}) = \min_{b \in \{0,1\}^9} t(r)((b, b^1, \ldots, b^{n+1})).$$

Here, we assume (w.l.o.g.) that the bit vector of $\sigma(r)$ is occupying the first 9 bits of the bit vector that is used to index $t(r)$. When computing $t(r+1)$ (associated to $B(r+1)$) from $t'(r)$, we will always check for consistency. More precisely, with $b = (b_{\swarrow}, b_{\leftarrow}, b_{\nwarrow}, b_{\uparrow}, b_{\nearrow}, b_{\rightarrow}, b_{\searrow}, b_{\downarrow}, q)$ and $\vec{b} = (b^1, \ldots, b^{n+1})$, $t(r+1)(\vec{b}, b) = t'(r)(\vec{b}) + q$, whenever $(\vec{b}, b)$ is consistent. That is, assuming consistency, the number of queens in the stored minimum solution is incremented by one if and only if the q-bit is set.

Hence, we can assume consistency of all tables by induction, based on starting with a consistent table for the first bag. Again, we refrain from giving all details here, but only mention that in the case when we assume $q = 1$ in the bit vector of the new vertex, i.e., when we put a queen on that square, then we are consistent only with those $(n+1) \times 9$-bit-vectors where we have $b_{\rightarrow} = 1$ for the 9-bit-vector corresponding to the left neighbouring square; similar conditions apply for other neighbours.

4.4. How to speed up our algorithm

Do we actually need all those bits as described above? This is quite an important issue, since reducing bits means that we might be able to reduce the basis of the exponential function c^n that basically determines the running time of the sketched path decomposition based algorithm. We are currently at $c = 2^9$ and hence we get an algorithm that is (much) worse than our previous one.

Saving the q-bit. Our first observation is that we need not save the information whether a queen is residing in a square or not. Namely, notice that $q = 1$ for square s implies that some future square to be processed that is say right neighbour of s will notice that it is dominated from the left; the same observation may be due to the fact that s itself is dominated from the left. A similar situation is true for all other neighbours of s. Hence, from the point of view of the neighbours not yet processed, $q = 1$ is treated the same way as if all four bits that describe the "past" (so, whether s is already dominated from one of those four directions) are set to one. We can easily incorporate this case when computing $t(r+1)$ from $t'(r)$, focussing on the newly added square: namely, if all bits indicating the past for this new square are one, then this might be (a) due to seeing all lines already dominated, or (b) possibly some but not all lines are dominated, and we put a queen on the new square. This means that for this particular table entry of $t(r+1)$, we compute the minimum of (1) the table entries of $t'(r)$ stemming from (a) and (2) one plus the minimum of the table entries of $t'(r)$ corresponding to (b). So, the information whether or not a queen is put onto a particular square is never explicitly stored.

Columns. Our path decomposition has the property that each bag contains vertices of two neighbouring columns. This means that in our current solution, the information whether one of those two columns is dominated by a vertex from the bag or not is stored in a very redundant fashion. Instead of the $b_{\uparrow}$- resp. $b_{\downarrow}$-bits, we can save the information whether the column is dominated from within the bag, and if so, we may also want to store in which row the dominating queen resides.

With the first two ideas, we have come down from $2^9 = 512$ to $2^6 = 64$ different pieces of information per square in the bag, yielding a table of size $\mathcal{O}(2^{6n})$ per bag.

Squeezing opposite bits. For the remaining three lines (two diagonals and the row), we still have two bits per line. However, notice that the case that a certain square is already dominated from the left and we still require domination from the right is not very sensible. So, three pieces of (mutual exclusive) information (trits) per line are sufficient, say 1 standing for "line is already dominated", 0 meaning "line not dominated (at all)" and -1 representing "line is still to be dominated". This means that we have come down from $4^3 = 64$ to $3^3 = 27$ different pieces of information per square in the bag.

Saving space. Similar to our comments following Theorem 2, we can save considerable space by fixing, in an outer loop of our algorithm, those rows which are directly dominated (by putting queens on those rows). This outer loop considers about 2^n different situations. So, when processing along the path decomposition, we only have to specify in which way the rows we

selected to be dominated are actually covered. For the overall running time, nothing has changed compared to our previous analysis, since at each stage of the path decomposition based algorithm, we consider all three situations of row domination, the only difference being the way this information is maintained. In terms of storage, instead of the row trit, we now have a row bit, whose information is only important when the outside loop decided to directly dominate a specific row. Hence, the table size shrinks down to $\mathcal{O}(18^n)$.

We can summarise our findings:

Theorem 3. MINIMUM DOMINATING SET OF QUEENS *can be solved in time* $\mathcal{O}(27^n p(n))$ *and in space* $\mathcal{O}(18^n)$ *given an* $n \times n$ *chessboard, where* p *is some polynomial.*

4.5. Didactic remarks and the sweep line interpretation

Treewidth-/pathwidth-based algorithms seem to be hard to explain to beginners. By way of contrast, geometrically motivated sweep line algorithms seem to be conceptually simpler. We explain in the following that this need not be the case. A sweep line usually serves as a separator of the underlying geometric structure. The sweep line moves "forward" in one direction, and in one forward step of the sweep line, the information associated to the sweep line has to be updated. How can this paradigm be applied to our case?

The sweep line would correspond to one column of the chessboard. It moves from left to right. Its state is a combination of the states of the squares that can be, in the simplest form, again described by a 9-bit-vector. The update of the sweep line state (upon moving the line to the right) is more involved than in the path decomposition case, since more cases must be considered. However, once the students understood that this update is not that easy to implement (although it can be quite easily explained in words), they should be more prepared to digest the intricateness of the path decomposition definition. Obviously, the sweep lines could be intercalated as an extra bag into the path decomposition itself, so that the former path decomposition bags would now play the role of auxiliary intermediate "states".

There is an interesting game-theoretic formulation of the notion of pathwidth, yielding the so-called *node search number* (which is known to be always one less than the pathwidth of a graph, see [21,29]): this gives the minimum number of searchers that are needed to chase down an invisible fugitive on a graph. All participants in such a search game may only be placed on vertices. However, the fugitive can move arbitrarily along edges, as long as he does not rush into searchers (if he does reside on the same vertex as a searcher, he would be caught), while searchers can move arbitrarily. Given any initial position of participants on the graph vertices, how many searchers are needed to always catch the fugitive? Notice that a sweepline corresponds to a row of searchers that move through the graph "in parallel", so this might even yield a further notion of search game, see [20] for a recent overview; possibly, "fast search" is a notion that comes closest to the line-sweep notion, see [14]. We believe that such game-theoretic formulations can be used to further illustrate the possibly tough-looking definition of pathwidth.

4.6. More on the pathwidth approach to chessboard problems

Further details on pathwidth and treewidth and according algorithms can be found in [1,5,4,15,22]. We also mention that it is rather easy to modify our sketched path decomposition based algorithm to obtain an $\mathcal{O}(8^n n^2)$ algorithm to compute the number of optimum solutions to the queen independence problem, matching the running time of the algorithm given in [28] that is based on dynamic programming on subsets. Namely, we only have to keep track of whether or not we already put a queen on a specific line.

So, all algorithmic techniques described in this paper can be exemplified with that problem, as well.

5. Complexity issues

Surprisingly little is known regarding the computational complexity of MINIMUM DOMINATING SET OF QUEENS; for example, it is unknown if the natural decision problem is $\mathcal{NP}$-hard, given n in unary. One can argue (in view of the relatively tight combinatorial bounds) that this is unlikely to achieve; however, it seems to be also difficult to produce a polynomial-time algorithm for this problem, keeping in mind our struggle to obtain single-exponential algorithms for the problem (measured against the side length of the board). This complexity question is somewhat embarrassing from a pedagogical point of view: if we teach both basics of programming and of complexity theory at undergraduate level, it is nagging that an answer is still pending to the seemingly simple question if a polynomial-time algorithm for MINIMUM DOMINATING SET OF QUEENS is achievable. After all, this might mean that this favourite backtracking example would need no backtracking at all. Having said this, one might also suspect that in some way $\mathcal{NP}$-hardness can be finally shown. Then, it would be also interesting to look at questions like approximability or fixed parameter tractability. This is what we will briefly do in this section. More details on both mentioned approaches can be found in [2,13,15].

Observe that, in particular through the papers of Cockayne, Burger and Mynhardt, there has been a sort of race for upper bounds on MINIMUM DOMINATING SET OF QUEENS. Since all corresponding constructions are in fact algorithmic, we can immediately conclude from our discussions in the introduction:

Corollary 1. *Ignoring additive constants,* MINIMUM DOMINATING SET OF QUEENS *can be approximated up to a factor of* $\frac{138}{133}$.

The standard way of parameterizing minimization problems is to choose the entity to be minimised as the parameter; in fact, this is the way we introduced DOMINATING SET OF QUEENS in Section 1. We need the following result essentially due to Raghavan and Venkatesan [26], and obviously independently by Cockayne [10].

Theorem 4. *Let k be the minimum number of queens needed to dominate an $n \times n$ chessboard. Then, $2k + 1 \geq n$.*

Theorem 4 justifies the following reduction rule:

Reduction rule 1. *If C is an $n \times n$ board and k a parameter with $n > 2k + 1$, then* NO.

Hence, a reduced $n \times n$ board satisfies $n \leq 2k + 1$; this shows fixed parameter tractability by reduction to a so-called problem kernel of size $(2k + 1)^2$ (measuring the overall size of the board). Accordingly, all algorithms we presented in the preceding sections can be also read as fixed parameter algorithms. More specifically, we can conclude:

Corollary 2. *A path decomposition based algorithm, when applied to a reduced* DOMINATING SET OF QUEENS *instance, runs in time $\mathcal{O}(27^{2k}p(k))$ for some polynomial p.*

6. Conclusions

We have presented three different ways of computing MINIMUM DOMINATING SET OF QUEENS. It would be interesting to see if the still menacing exponential bases of the single-exponential algorithms can be further brought down. In particular, it would be very nice if the theoretically better $\mathcal{O}(c^n)$ algorithms would really "beat" the branch-and-bound algorithms in practice. Notice that our $\mathcal{O}(c^n)$ algorithms need exponential space, while the simple backtracking algorithm only takes polynomial space, which brings in the question to present a single-exponential algorithm that only consumes polynomial space.

Coming back to our original question: Is MINIMUM DOMINATING SET OF QUEENS a trivial programming exercise? We have shown that this is not the case, to the contrary, it is quite a challenging algorithmic problem, which can be used to exhibit some of the most advanced techniques available for solving hard combinatorial problems. According to our computer experiments, the most efficient way to generate small solutions seems to even be a randomised approach. In this context, also some basics of linear programming might be explained, according to the lines of Ref. [9].

One of the most intriguing open problems, however, is already mentioned in [26]: Is there an easy computational procedure to solve MINIMUM DOMINATING SET OF QUEENS? Similar problems that can be discussed refer to total domination or independent domination; see [11,26] for some combinatorial bounds on these problems. Notice that all our algorithmic approaches can be readily adapted to cope with these variants.

Acknowledgements

I gratefully acknowledges lots of constructive comments from my student Sebastian Schlecht, who also pointed out some problems with an earlier version of my path decomposition based algorithm that led to a claimed smaller base of the exponential run time estimate in the CTW conference version of this paper. Moreover, the constructive comments of the referees are gratefully acknowledged.

References

[1] S. Arnborg, J. Lagergren, D. Seese, Easy problems for tree-decomposable graphs, Journal of Algorithms 12 (1991) 308–340.
[2] G. Ausiello, P. Creczenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, M. Protasi, Complexity and Approximation; Combinatorial Optimization Problems and Their Approximability Properties, Springer, 1999.
[3] J.-P. Bode, H. Harborth, Independence for knights on hexagon and triangle boards, Discrete Mathematics 272 (2003) 27–35.
[4] H.L. Bodlaender, Discovering treewidth, in: P. Vojtáš, M. Bieliková, B. Charron-Bost, O. Sýkora (Eds.), SOFSEM, in: LNCS, vol. 3381, Springer, 2005, pp. 1–16.
[5] H.L. Bodlaender, A partial k-arboretum of graphs with bounded treewidth, Theoretical Computer Science 209 (1998) 1–45.
[6] H.L. Bodlaender, F.V. Fomin, A.M.C.A. Koster, D. Kratsch, D.M. Thilikos, On exact algorithms for treewidth. Technical Report UU-CS-2006-032, Department of Information and Computing Sciences, Utrecht University, 2006. Long version of the ESA 2006 abstract.
[7] A.P. Burger, C.M. Mynhardt, An upper bound for the minimum number of queens covering the $n \times n$ chessboard, Discrete Applied Mathematics 121 (2002) 51–60.
[8] A.P. Burger, C.M. Mynhardt, An improved upper bound for queens domination numbers, Discrete Mathematics 266 (2003) 119–131.
[9] M.J. Chlond, IP modeling of chessboard placements and related puzzles, INFORMS Transactions on Education 2 (2) (2002) 56–57.
[10] E.J. Cockayne, Chessboard domination problems, Discrete Mathematics 86 (1990) 13–20.
[11] E.J. Cockayne, P.H. Spencer, On the independent queens covering problem, Graphs and Combinatorics 4 (1988) 101–110.
[12] F. Dorn, E. Penninkx, H.L. Bodlaender, F.V. Fomin, Efficient exact algorithms on planar graphs: Exploiting sphere cut branch decompositions, in: European Symposium on Algorithms ESA, in: LNCS, vol. 3669, 2005, pp. 95–106.
[13] R.G. Downey, M.R. Fellows, Parameterized Complexity, Springer, 1999.
[14] D. Dyer, B. Yang, Ö. Yaşar, On the fast searching problem, in: R. Fleischer, J. Xu (Eds.), Algorithmic Aspects in Information and Management AAIM 2008, in: LNCS, vol. 5034, Springer, 2008, pp. 143–154.
[15] H. Fernau, Parameterized Algorithmics: A Graph-Theoretic Approach. Habilitationsschrift, Universität Tübingen, Germany, 2005.

[16] D. Finozhenok, W.D. Weakley, An improved lower bound for domination numbers of the queen's graph, Australasian Journal of Combinatorics 37 (2007) 295–300.
[17] D.K. Garnick, N.A. Nieuwejaar, Total domination of the $m \times n$ chessboard by kings, crosses, and knights, Ars Combinatoria 41 (1995) 45–56.
[18] C.F. Gauß, Werke, vol. 12, Julius Springer, Berlin, 1929.
[19] R.K. Guy, Unsolved Problems in Number Theory, 2nd ed., Springer, 1994.
[20] P. Heggernes, R. Mihai, Mixed search number of permutation graphs, in: F.P. Preparata, X. Wu, J. Yin (Eds.), Frontiers in Algorithmics FAW, in: LNCS, vol. 5059, Springer, 2008, pp. 196–207.
[21] N. Kinnersley, The vertex separation number of a graph equals its pathwidth, Information Processing Letters 142 (1992) 345–350.
[22] T. Kloks, Treewidth. Computations and Approximations, in: LNCS, vol. 842, Springer, 1994.
[23] D.E. Knuth, Dancing links, in: Millenial Perspectives in Computer Science, Houndmills, Basingstoke, Hempshire, Palgrave, 2000, pp. 187–214.
[24] C. Letavec, J. Ruggiero, The *n*-queens problem, INFORMS Transactions on Education 2 (3) (2002) 101–103.
[25] P.R.J. Östergård, W.D. Weakley, Values of domination numbers of the queen's graph, Electronic Journal on Combinatorics 8:#R29 (2001) 19.
[26] V. Raghavan, S.M. Venkatesan, Bounds for a board covering problem, Information Processing Letters 25 (1987) 281–284.
[27] M. Reichling, A simplified solution of the *n* queens' problem, Information Processing Letters 25 (1987) 253–255.
[28] I. Rivin, R. Zabih, A dynamic programming solution to the *n*-queens problem, Information Processing Letters 41 (1992) 253–256.
[29] A. Takahashi, S. Ueno, Y. Kajitani, Mixed searching and proper-path-width, Theoretical Computer Science 137 (2) (1995) 253–268.
[30] W.F.D. Theron, G. Geldenhuys, Domination by queens on a square beehive, Discrete Mathematics 178 (1998) 213–220.