Algorithm Performance For Chessboard Separation Problems

R. Douglas Chatham* Maureen Doyle† John J. Miller†‡
Amber M. Rogers†‡ R. Duane Skaggs* Jeffrey A. Ward†

April 23, 2008

Abstract

Chessboard separation problems are modifications to classic chessboard problems, such as the N Queens Problem, in which obstacles are placed on the chessboard. This paper focuses on a variation known as the $N + k$ Queens Problem, in which k Pawns and $N + k$ mutually non-attacking Queens are to be placed on an N-by-N chessboard. Results are presented from performance studies examining the efficiency of sequential and parallel programs that count the number of solutions to the $N + k$ Queens Problem using traditional backtracking and dancing links. The use of Stochastic Local Search for determining existence of solutions is also presented. In addition, preliminary results are given for a similar problem, the $N + k$ Amazons.

1 Introduction

According to the standard rules of chess, a queen can move any number of squares in a straight line vertically, horizontally, or diagonally, as long as no other piece lies in its path. Questions involving various placements of multiple queens on chessboards were first posed in the mid-19th century. In 1848, Max Bezzel created the 8 Queens Problem, which calls for a placement of eight queens on an 8×8 chessboard so that no two queens "attack" each other (i.e. no queen lies in another queen's path) [1].

*Department of Mathematics and Computer Science, Morehead State University, Morehead, Kentucky 40351 USA

†Department of Computer Science, Northern Kentucky University, Highland Heights, Kentucky 41099 USA

‡Undergraduate student

The 8 Queens Problem and several variations appear extensively through the mathematics and computer science literatures. (One such variation is the "N Queens Problem" which calls for placing N mutually non-attacking queens on an $N \times N$ board.) In mathematics, the problem has been connected to topics such as graph-theoretic domination, integer programming, and magic squares. In computer science, the problem is used as a model for backtracking programming techniques (including the dancing links method popularized by Knuth in [8]), constraint programming, parallel programming, and neural nets. A collection of references to the 8 Queens Problem can be found in [9]. We also refer the interested reader to [6] and [14].

In January 2004, the Chess Variant Pages [2] proposed a variation of the traditional 8 Queens Problem. The new problem, posed as part of a contest on the site, was to place nine queens on an 8×8 board by using the least number of pawns possible in order to block all queens that would otherwise attack each other. The contest winner was able to place nine queens with only one pawn, which immediately suggests a generalization to square boards of arbitrary order N with $N + k$ queens, where $k \geq 1$ is an integer. The "$N + k$ Queens Problem" is the problem of placing $N + k$ queens and k pawns on an $N \times N$ board so that no two queens attack each other. It was conjectured in [4] and proven in [3] that for each $k \geq 0$, for large enough N, the $N + k$ Queens Problem has at least one solution. In this paper we consider algorithms that count the number of solutions to the $N + k$ Queens Problem for various values of N and k.

We examine and present results for solving the $N + k$ Queens Problem using recursive backtracking and dancing links using a single processor. Dancing links is then modified to run on a multiprocessor Beowulf-like cluster and additional results are presented.

We also discuss two variants of this problem. A Stochastic Local Search algorithm, and its results, are presented for finding a solution to the $N + k$ Queens Problem in Section 3.

Finally, we discuss an approach to solving the $N + k$ Amazons Problem and present initial results in Section 4. The $N + k$ Amazons Problem is a modification to the $N + k$ Queens Problem where the piece can move as either a queen or a knight.

2 $N + k$ Queens

Finding a closed form expression for the number of solutions to either the N Queens or the $N + k$ Queens Problem seems highly unlikely. Asymptotic results have been given for the N Queens Problem [12] and numerous algorithms have been proposed for counting the number of solutions.

Two different exhaustive search methods, recursive backtracking and dancing links, for solving the $N + k$ Queens Problem were examined in [3]. We provide improved versions of these two methods in Section 2.1. Stochastic Local Search is discussed in Section 3 as an approach to finding single solutions to the $N + k$ Queens Problem.

2.1 Exhaustive Search

In [3], solutions to the $N + k$ Queens Problem based on traditional recursive backtracking and dancing links were considered and compared, but not optimized. Backtracking was implemented as a standard backtracking algorithm, placing a queen in each row and proceeding. Dancing links, discussed in the following section, was used to solve the $N + k$ chessboard given a valid pawn placement. However, the valid pawn placements were computed simply using traditional backtracking as part of the algorithm initialization. In this paper, we describe a process of using nested dancing links and optimizations for pawn placement and row selection in order to improve execution times for both algorithms.

An immediate way to improve execution times for exhaustive searches is to detect faulty solutions as early as possible. Proposition 13 from [3] and Corollary 1 provide useful criteria for pruning.

Proposition 13 ([3]) *If $N + k$ queens and k pawns are placed on an $N \times N$ board so that no two queens attack each other, then no pawn can be on the first or last row, first or last column, or any square adjacent to a corner.*

Corollary 1 *If $N + k$ queens and k pawns are placed on an $N \times N$ board so that no two queens attack each other, then no two pawns can be adjacent to each other horizontally or vertically.*

Proof. Suppose there is a row or column with p pawns such that two of the pawns are adjacent. A queen can not be placed between the two adjacent pawns, therefore the row or column can be divided into at most p parts.

Suppose there are p queens in a row with p pawns, two of which are adjacent. There are $k-p$ pawns in the other rows, and at most $k-p+N-1$ queens can be placed in those rows. For this arrangement, $p+k-p+N-1 = N + k - 1$ queens can be placed. This contradicts the given that the board has $N + k$ queens placed on it. ■

2.1.1 Backtracking

Typical backtracking solutions for the N Queens Problem use N recursive calls, placing a queen in row i for each recursive call i. For the $N+k$ Queens

Problem, there are $N+2k$ recursive calls. The first k calls place k pawns on the chessboard, dividing the rows and columns of the chessboard into $N+k$ row segments and $N+k$ column segments. The remaining $N+k$ calls recursively place a queen on a segment. Figure 1 illustrates the row segment assignment for a pawn placement when $k=2, N=5$.

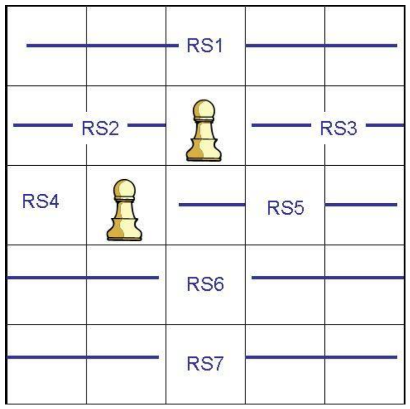

Figure 1: $N+k$ Row Segments

The initial backtracking implementation made $N+k$ recursive calls for each row segment. A queen was placed in each valid column for the row segment. The row or row segment selection was the i^{th} row or row segment for the i^{th} recursive call.

The new row selection criteria is a modification of Knuth's "organ pipe ordering" defined in [8]. Instead of placing a queen in the i^{th} row for the i^{th} recursive call, each recursive call selects the most constrained row **or** column. The most constrained row or column is defined as the row or column with the fewest squares available for placing a queen. Differing from Knuth in the case of a tie, both of our implementations of backtracking and dancing links simply select the first encountered row or column. The general algorithm is given in Figure 2.

2.1.2 Dancing Links

Dancing links was introduced in 1979 by Hitotumatu and Noshita [7] and popularized by Knuth in 2000 [8]. Knuth provides an excellent explanation of the Dancing Links algorithm, which is a technique to implement Knuth's Algorithm X to solve exact cover problems. We provide an abbreviated explanation here for completeness.

```
void backtrack(int i, int totalQueens)

  if (i equals totalQueens)
     incrementTotalSolutions()
     return

  if (!constrainedRowFound(row))
     return

  for (int j = firstCol(row); j <= lastCol(row); j++)
      placeQueen(i, j)
      backtrack(i+1, totalQueens)
      removeQueen(i, j)
```

Figure 2: $N + k$ Backtracking Algorithm

Dancing links is a technique involving a quadruply linked-list data structure and an exhaustive search algorithm. When using dancing links to solve chessboard problems, a header node is connected to $6N - 2$ column header nodes. Each column header node represents a unique row (N), column(N), lower diagonal($2N - 1$) or upper diagonal($2N - 1$) from the chessboard.

Each column of the DLX structure contains its column header node and also one node for each chessboard block in that row, column or diagonal. Therefore, there are N nodes in the DLX column for the chessboard row and columns. There are between 1 and N nodes in the columns representing the diagonals.

Finally, each chessboard block is represented by four horizontally connected nodes, one for the blocks row, one for its column, one for its upper diagonal and one for its lower diagonal. Figure 3 shows a two-by-two chessboard and its dancing links data structure.

Knuth defines the algorithm with three methods solve, cover, and uncover. The main method, solve, is called recursively, once for each queen when solving the N Queens Problem. solve chooses a DLX structure column for placing a queen and iterates through all queen positions (nodes) available in the column, calling itself recursively for valid queen placements. The recursion halts and returns when a solution is found, or all positions have been tried.

Based on the initial success of dancing links in [3], the use of nested dancing links was explored. In [3], $N + k$ Queens was solved by first computing all valid pawn placements, and then solving the resulting data structure.

Modifications to the dancing links data structure is minimal when ap-

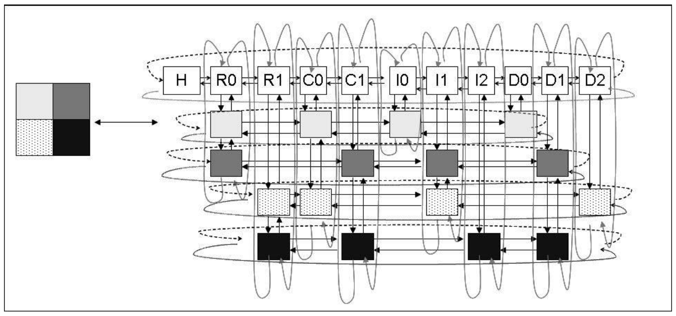

Figure 3: 2 × 2 Chessboard and Data Structure

plying it to the $N+k$ Queens Problem. Each valid pawn placement divides up to four columns in the DLX structure: a chessboard row, chessboard column, and up to two diagonal columns. The pawn can not attack, so no covering nor uncovering of additional nodes is required. Once all k pawns are placed, solve is called to place $N+k$ queens.

The placing of the pawns requires three new methods nkQueens, placePawn, and removePawn and no changes to existing N Queens code.

The main method, nkQueens, is called with nkQueens(0,k,N, 0, 0) and is defined in Figure 4. nkQueens is recursively called k times to place the k pawns. Once the pawns have been placed, this method calls solve to place the queens. For simplification, this algorithm does not include either of the pruning criteria established in Section 2.1.

2.2 Results

The parallel results were generated using a 16-node, Pentium D 2.8 GHz Beowulf-like cluster with gigabit Ethernet. The head node has 2GB of RAM and the other nodes each have 1GB of RAM. The cluster runs the Fedora Core 5 operating system and utilizes the Open Source Cluster Application Resources (OSCAR) software package [11]. The implementation was in C++ using Message Passing Interface (MPI).

Initial results for comparing backtracking and dancing links were done using the head node of the cluster.

2.2.1 Sequential

Two sequential approaches were first examined for solving the $N+k$ Queens Problem. The sequential implementations were run on a single-processor PC. Table 1 contains the timing results for the $N+k$ Queens problem, varying N and k, for the two solvers. Each timing result is the average at least five runs of the algorithm for the specific combination of (N,k). For

```
void nkQueens(int i, int totalPawns, int totalQueens, int row, int col)

  if (i == totalPawns)
     solve(0, totalQueens+totalPawns)    // calls N Queens solver
     return

  // Place this pawn
  if (col > totalQueens-1)
    increment row by 1

  // Consider all additional positions for this pawn
  for ( col = (col + 1) % totalQueens;
        (col < totalQueens) && (row <  totalQueens);
        col = (col + 1) % totalQueens)
       placePawn(i,row, col )
       nkQueens(i+1, totalPawns, totalQueens)
       removePawn(row, col, i)
       if (col > totalQueens - 1)
         increment row by 1
```

Figure 4: $N + k$ Queens DLX Solver

run times larger than one second, the fastest average time is highlighted in **bold**.

Table 1: Sequential $N + k$ Queens, timing

N/k	Alg	1	2	3	4	5
6	BT	0.328	0.7486	0.187	0.249	0.166
	DLX	0.226	0.488	0.2789	0.109	0.310
7	BT	0.126	0.332	0.141	0.437	0.877
	DLX	0.439	0.176	0.136	0.847	4.237
8	BT	0.557	0.137	0.738	**3.780**	**14.197**
	DLX	0.211	0.613	0.553	4.681	32.504
9	BT	0.266	0.693	3.364	21.849	**126.586**
	DLX	0.726	0.222	**1.892**	**20.634**	190.952
10	BT	1.469	4.761	17.999	110.284	**813.451**
	DLX	0.391	0.842	**6.228**	**83.950**	975.214
11	BT	8.693	29.503	111.939	595.766	4438.234
	DLX	2.319	5.342	**27.843**	**288.883**	**3931.816**

Table 1 shows that if both backtracking and DLX use the “organ pipe ordering” then backtracking is sometimes faster for smaller N. However, as N increases, the initial overhead required to create the DLX data structure is not so predominant and the faster algorithm is DLX by an order of magnitude.

2.2.2 Parallel

As shown in the previous section, DLX is more efficient than backtracking. As a result of this, only DLX was modified to solve the $N+k$ Queens Problem in parallel using a Beowulf-like cluster. The parallel version of DLX parcels the problem out to different processors based on the positions of the first pawn. By ordering the chessboard ascending by row then by column, and enforcing the rule that no pawn is permitted to be placed adjacent to another pawn horizontally or vertically, we are able to ensure that we do not duplicate any pawn positions. Each processor is initially assigned one unique starting pawn position. The first pawn is placed and the algorithm proceeds solving the $N+k$ Queens Problem using the sequential algorithm. Once all possible pawn combinations are tried, given the starting pawn position, the processor is assigned another starting pawn position.

We examined dividing starting pawn positions by segmenting the chessboard, but observed for small N that the work was not evenly distributed among the processors. To mitigate this, a round-robin approach is used instead. Given P is the total number of processors, p is the processor identifier, where $0 \leq p < P$, and N is the number of queens, then the starting row and column $(r_p(i), c_p(i))$ for each pawn, i, is computed by

$$r_p(i) = \begin{cases} (P * x + N + p)\%(N-2) & i \text{ even} \\ (P * (x-1) + N + (P-p-1) + P)\%(N-2) & i \text{ odd} \end{cases}$$

$$c_p(i) = \begin{cases} (P * x + N + p)/(N-2) & i \text{ even} \\ (P * (x-1) + N + (P-p-1) + P)/(N-2) & i \text{ odd.} \end{cases}$$

Table 2 presents the total solutions to the $N+k$ Queens Problem solved. Fundamental solutiosn are presented in Table 3, where the set of fundamental solutions of a chessboard problem is the set of solutions such that no solution is a rotation or reflection of another.

The average execution times for five runs of each (N, k) combination are shown in Table 4. This parallel implementation has an average speedup of 5.3 and a maximum speedup of 17.5 ($N = 11, k = 3$). The average speedup is much less than the ideal speedup of 32 indicating that additional speedup is possible.

The timing results show, as expected, an almost order-of-magnitude increase between N and $N+1$ holding k constant when $N > 11$. Similarly,

Table 2: Parallel $N + k$ Queens, Total Solutions

N/k	1	2	3	4	5
6	16	0	0	0	0
7	20	4	0	0	0
8	128	44	8	0	0
9	396	280	44	8	0
10	2288	1304	528	88	0
11	11152	12452	5976	1688	196
12	65712	10512	77896	30936	7032
13	437848	977664	1052884	627916	225884
14	3118664	9239816	13666360	11546884	6077320
15	23387448				
16	183463680				

Table 3: Parallel $N + k$ Queens, Fundamental Solutions

N/k	1	2	3	4	5
6	2	0	0	0	0
7	3	1	0	0	0
8	16	6	1	0	0
9	52	37	6	1	0
10	286	164	66	11	0
11	1403	1572	751	215	29
12	8216	13133	9737	3871	879
13	54756	122279	131672	78560	28268
14	389833	1155103	1708295	1443461	759665
15	2923757				
16	22932960				

we observe the same exponential increase between k and $k+1$, when $k > 2$ and $N > 6$.

3 Stochastic Local Search

Stochastic Local Search (SLS) is an approach that is often useful for solving hard combinatorial problems. In general, SLS involves first constructing an initial state in which all of the variables in the problem are assigned values,

Table 4: Parallel $N + k$ Queens, timing

N/k	1	2	3	4	5
6	0.801	0.1548	0.103	0.422	0.206
7	0.225	0.268	0.660	0.265	1.457
8	0.489	0.470	0.805	0.641	6.842
9	0.148	0.120	0.259	3.341	44.601
10	0.306	0.787	0.815	7.795	107.795
11	0.133	0.408	1.588	20.447	358.705
12	0.548	2.123	7.630	74.567	1197.706
13	4.996	22.403	55.276	290.448	4900.838
14	22.419	115.352	366.267	1579.269	16757.187
15	215.151				
16	2799.529				

and then repeatedly modifying the current state into a "nearby" state until a solution is reached. Some combination of randomization and heuristics is used at each point in the search in order to determine which state to visit from the current state. Commonly, SLS algorithms will "restart" from a new initial state after a certain amount of time if the current search path has not been fruitful. SLS algorithms are usually *incomplete* in that they do not have the capability to report whether a problem is unsolvable, nor can they enumerate all of the solutions to a problem. However, in many domains they have the capability to solve large combinatorial problems that are intractable via complete procedures such as backtracking.

SLS algorithms have been very successful in solving large instances of the N Queens Problem. For instance, Sosič and Gu [13] present an SLS algorithm that runs in linear time and is able to find solutions for very large instances, such as solving the 3,000,000 Queens Problem in under five seconds on a current workstation.

Sosič and Gu's algorithm consists of two phases: an Initial_Search phase which constructs the initial search state, and a Final_Search phase which attempts to transform the current state into a solution.

The Initial_Search proceeds left to right across the columns of the board. At each column index, c, a trial row index r_c is chosen randomly from the set of rows that are not yet occupied, until a value for r_c is found such that placing a queen at column c and row r_c will not conflict (i.e. along any diagonal) with a queen that has already been placed. Initial_Search will perform a total of no more than $3.08N$ trials across all of the columns of the board. If the limit of $3.08N$ trials becomes exhausted at some column

c', then the values $r_{c'}, r_{c'+1}, ..., r_N$ are chosen to be a random permutation of the remaining available row numbers, regardless of any diagonal conflicts that may exist with these choices. Sosič and Gu state that an aim of the Initial_Search procedure is to make c' as close as possible to N, in preparation for Final_Search. This objective is successfully achieved to a very great degree, such as a value of $c' = 2{,}999{,}977$ that was obtained when we made a trial run of their algorithm to solve the 3,000,000 Queens Problem.

Sosič and Gu's Final_Search method works from left to right across the remainder of the board, with $c = c', ..., N$. For each value of c, a column d is chosen at random repeatedly until a d is found such that swapping the row values in columns c and d results in no conflicts for the queens in either of those columns. Once this has been completed for $c = N$, then a solution to the N Queens Problem has been found. If Final_Search considers a total (across $c = c', ..., N$ of 7000 trial values for d without finding a solution, then the algorithm restarts, calling Initial_Search to reinitialize the current state.

An important invariant that is in effect in Sosič and Gu's algorithm after the construction of the initial state is that every state that is explored has exactly one queen in each row and exactly one queen in each column.

In designing an SLS algorithm for the $N + k$ Queens Problem, we used Sosič and Gu's idea of having an initialSearch method that seeks to construct an initial state with relatively few conflicts, followed by a finalSearch method. However, the k pawns present a significant problem in adapting the remainder of the Sosič and Gu approach. For example, an obvious counterpart to their invariant would be to maintain an invariant that states that every column segment and every row segment has exactly one queen in it. However, then, in the general case, swapping row values between two columns could break the invariant, both with respect to the column segments and the row segments. In Sosič and Gu's algorithm, swapping row values never breaks the invariant that every row and every column contains exactly one queen.

The invariant that we chose to maintain is that every column segment has exactly one queen in it. Thus, our local search method has to deal with both row conflicts and diagonal conflicts as the search proceeds. Prior to calling initialSearch, our algorithm randomly places k pawns on the board subject to the restrictions that no pawn may be on the edge of the board, or adjacent to a corner square, and no two pawns may be adjacent to each other along a column or a row. Our initialSearch method iterates through the columns from left to right. For each column segment in a column, we randomly probe up to 10,000 times the row indices that are available in the segment, keeping track of those row indices that result in the minimal number of conflicts (i.e. row conflicts plus diagonal conflicts) with queens that have already been placed. If a probe yields a row position that results in no conflicts, then a queen is immediately placed in the current column segment, at that row. Otherwise, at the end of the 10,000 probes, we randomly choose one of the row indicies that resulted in the minimal number of conflicts, placing a queen in the current column segment at that row.

finalSearch uses a "hill-climbing" approach that seeks to minimize the total number of conflicts that exist along row segments and diagonal segments. finalSearch keeps a conflict list of all column segments that have queens with at least one conflict. Each move made by finalSearch consists of moving a single queen to a different square on its column segment. The selection of a move involves repeatedly traversing the *conflict list* up to 100 times. Each time that an entry is visited on the list, a random square is selected from the column segment. If moving to that square would result in a reduction in the number conflicts for this queen, then the move is made. If not, then the algorithm notes the amount by which this move increases the total number of conflicts. (Call this the *score* of the move, which may be zero.) If no move has been made after 100 traversals of the conflict list, then, from the set of moves that were found to have the best (lowest) scores, a move is randomly chosen and performed. Occasionally, the best score obtained is positive, in which case the move actually increases the total number of conflicts. But such moves can serve to move the current state out of a local minimum.

If, after a finalSearch move, the conflict list has a size of only two (i.e. there is only one pair of conflicting queens remaining) then finalSearch considers the first queen on the conflict list and performs an exhaustive search of that queen's column segment to determine whether there an available square where that queen could be moved to immediately solve the puzzle.

We have not yet found restarting to be useful for our algorithm, although this will be an area for future work.

Table 5 shows run times obtained from this SLS algorithm in solving $N + k$ Queens problems. While the algorithm is not able to enumerate all solutions to a problem instance, or to determine that a problem has no solutions, it does extend by orders of magnitude the sizes of $N + k$ Queens problems for which we can obtain a solution.

Table 5: $N + k$ Queens SLS, timing

N/k	1	10	100	1000
10000	0.43	0.78	5.38	600.83
100000	9.61	10.48	54.84	799.20
1000000	39.73	54.67	248.98	>3600

These runtimes were obtained on a 3GHz Pentium 4 with 1.5GB RAM running Windows XP.

4 Amazons

In chess, a knight is a piece that can leap from one corner of a 2×3 rectangle to the opposite corner (i.e., two squares vertically or horizontally followed by one square in an orthogonal direction). In some chess variant games, an amazon is a piece that can move as either a queen or a knight. We consider the $N + k$ Amazons Problem – given an $N \times N$ board, place k pawns and $N + k$ mutually nonattacking amazons – since it is an easy variation of the $N + k$ Queens Problem. Since an amazon can move as a queen, a solution to the $N+k$ Amazons Problem is also a solution to the corresponding $N+k$ Queens Problem (just replace the amazons by queens). So, for example, an $N + k$ Amazons solver can use Proposition 13 and Corollary 1 for pruning in an exhaustive search.

We conjecture that the $N + k$ Amazons problem has solutions for large enough boards:

Conjecture 1 *For every $k \geq 0$, for large enough N, it is possible to place k pawns and $N+k$ mutually nonattacking amazons on an $N \times N$ board.*

As a result of the promising results using DLX for the $N + k$ Queens Problem, the $N + k$ Queens algorithm was modified to solve for $N + k$ Amazons. Two additional methods were added: setKnightMove and unsetKnightMove. When an amazon is placed in solve, setKnightMove is called to remove the chessboard squares that would be accessible from a knight move. When the amazon is removed, unsetKnightMove returns the chessboard squares to the data structure.

Since the amazon solver extends the $N + k$ queens implementation, nkQueens, it automatically takes advantage of the optimizations implemented for the $N + k$ Queens Problems described in Proposition 13 and Corollary 1.

The total solutions are given in Table 6 and the timing results for the Amazons $N + k$ Problem presented in Table 7. These results also demonstrate that Conjecture 1 is true for at least a finite set of N and k.

5 Conclusions and Future Work

The use of nesting the dancing links algorithms, applying it first for pawn placement and then for queen placement, allowed us to obtain additional

Table 6: Total $N+k$ Amazons solutions

N	0	1	2	3	4
9	0	0	0	0	0
10	4	0	0	0	0
11	44	0	0	0	0
12	156	72	0	0	0
13	1876	412	120	0	0
14	5180	10320	1664		
15	32516	71212			
16	202900				
17	1330622				

Table 7: Parallel $N+k$ Amazons, timing

N	0	1	2	3	4
10	0.016	0.063	0.594	8.782	132.11
11	0.062	0.281	1.89	25.047	401.456
12	0.312	1.641	8.704	74.969	1208.65
13	1.719	10.954	55.641	302.049	3711.2
14	8.64	8 0.625	437.003	1956.79	
15	49.907	597.894			
16	302.892				
17	1925.23				

solutions for the $N+k$ Queens Problem as compared to the results presented in [4] and [3]. This solver also provided us a framework to obtain solutions for the $N+k$ Amazons Problem.

Given the results presented here, there are many possible avenues to pursue. The authors are interested in exploring what symmetries in the $N+k$ Queens Problem exist, and how those symmetries impact the solvers. We also plan to examine the $N+k$ Queens Problem in three dimensions and consider the $N+k$ Queens Problem on a torus using dancing links. Future work might also include improving the speedup observed for the $N+k$ Queens solver and finding additional applications for these solvers.

Finally, constraint programming systems such as ECLiPSe [5] and Oz/Mozart [10] provide concise, declarative means for expressing problems such as N Queens. (Furthermore, Oz/Mozart provides built-in support for parallel processing.) We are interested in exploring the degree to which these highlevel languages can support $N + k$ Queens programming solutions which are both concise and efficient.

6 Acknowledgements

This work was funded by Kentucky NSF EPSCoR grant UKRF 3046884400-07-419. The authors thank the referee for helpful comments and for indicating potential improvements in the results through the use of ECLiPSe or Oz/Mozart.

References

[1] M. Bezzel, *Berliner Schachzeitung*, 3 (1848), 363.

[2] W.H. Bodlaender, *http://www.chessvariants.org/problems.dir/9queens.html*, January – March 2004.

[3] R.D. Chatham, M. Doyle, G.H. Fricke, J. Reitmann, R.D. Skaggs, and M.Wolff, Independence and domination separation on chessboard graphs, to appear in *J. Combin. Math. Combin. Comput.* (2008).

[4] R.D. Chatham, G.H. Fricke, and R.D. Skaggs, The queens separation problem, *Util. Math. 69* (2006), 129-141.

[5] ECLiPSe *http://eclipse.crosscoreop.com/*

[6] C. Erbas, S. Sarkeshik, and M. M. Tanik, Different perspectives of the N-queens problem, in *Proceedings of the ACM 1992 Computer Science Conference.*

[7] H. Hitotumatu and K. Noshita, A technique for implementing backtrack algorithms and its application, *Inform. Proc. Lett. 8* (4) (1979), 174-175.

[8] D.E. Knuth, Dancing links, in *Millenial Perspectives in Computer Science: Proceedings of the 1999 Oxford-Microsoft Symposium in Honour of Sir Tony Hoare*, J. Davies, A.W. Roscoe, and J. Woodcock eds., Palgrave (2000).

[9] W. Kosters, *http://www.liacs.nl/home/kosters/nqueens.html* (April 2002).

[10] Mozart Programming System *http://www.mozart-oz.org/*

[11] OSCAR *http://oscar.openclustergroup.org/.*

[12] I. Rivin, I. Vardi, and P. Zimmermann, The n-queens problem, *Amer. Math. Monthly* **101** (1994), 629-639.

[13] R. Sosič and J. Gu, Efficient local search with conflict minimization: A case study of the N-Queens problem, *IEEE Transactions on Knowledge and Data Engineering 6* (5) (1994), 661-668.

[14] J.J. Watkins, *Across the Board: The Mathematics of Chessboard Problems*, Princeton University Press (2004).