# Artificial Intelligence 1
# Lab 2

Robbin de Groot (s3376508) & Ethan Waterink (s3417611) & Lennard Manuel (s349952)
CS4/LC8_team[public_static_void_main()]

18 May 2018

## Exercise 1: N-queens problem

### 1.1 - Hill Climbing

1. We ran the program multiple times. The results are for 8 and 50 queens are in the appendix at the end of the document. We notice that, the larger the grid, the fewer queens are under attack. However, the program almost never solves the problem, but often gets close.

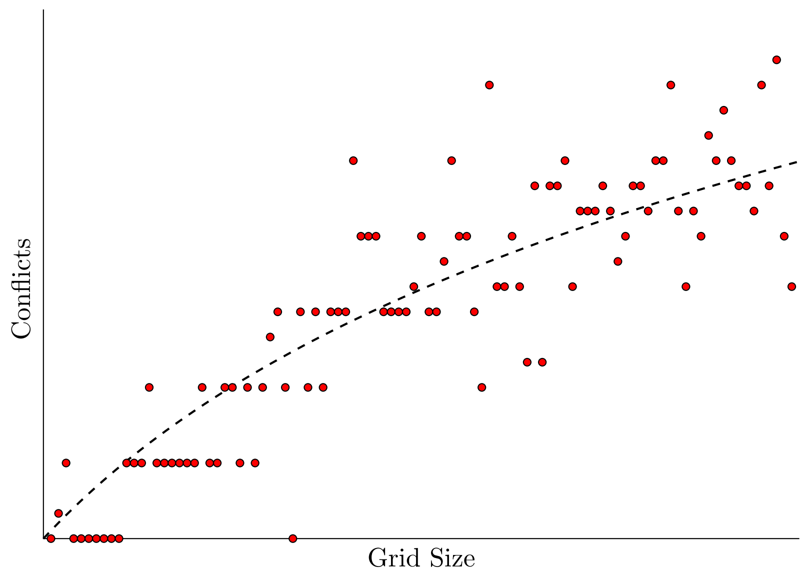
2. The algorithm loops over all queens, and puts them each in the most favourable position. This way, it might look like the solution is going to be optimal in the end, but the pseudo-optimal placement of the first few queens might hinder the yet-to-be-placed queens. Therefore, if this situation occurs, the found 'solution' might not actually be the optimal one, thus the program fails.

3. We might improve the algorithm by implementing a so called Random-Restart-Hill-Climbing algorithm. The idea behind this is as follows. If we dont find the solution, we try again, from another random point. The problem with this approach regarding this problem, is that, there are a lot of random restarts possible. Therefore, we must set a cap to the amount of restarts we allow, say, 25.
   We ran a test, and immediately got a solution:

```
Final State
.....q..
..q.....
....q...
.......q
q.......
...q....
.q......
......q.
```

This is plenty of confirmation that this is a viable solution.

4. We use a bash script to run the program 3 times for every possible value for `N`. We can then graph the average of the results for every grid size (along the horizontal axis).




## 1.2 - Simulated Annealing

1. For the most part, we copied the algorithm from the textbook. However, instead of using the loop for $t = 1$ to infinity, we used a nested loop, the outer one going from $tries = 0$ to $tries = 1000$, and the inner one going from $i = 0$ to $i = nqueens$. In this inner loop, the queen of every row is evaluated. If the random next position, called $newPos$ has less conflicts than the current position, the queen is moved to this $newPos$. If it has more (or the same amount of) conflicts than the current position, we generate a random number $r$ and if the probability $p = e^{(deltaE/T)}$ is bigger than this random float, we move the queen to the $newPos$ anyway, even though it has more conflicts. We do this the queen of every row, and try it 500 times, which is the outer loop. Instead of using the delta E from the psuedo-code, we use $deltaE = current.conflicts - next.conflicts$, which is exactly the same, just using one function less. One thing we added to improve the program was the same idea we used in question 3 from Hill Climbing. Basically, if $t$ is smaller than a very small value like 0.001 and a solution has not been found yet, we restart the program, however on a new random board. This is implemented by a new outer loop which goes from 0 to, in this case, 50. In this outer loop, the temperature is set to the start temperature again and the queens are initiated by InitiateQueens(1)

(which sets the queens on the board randomly).

2. We chose this as the function timeToTemperature:

```
int timeToTemperature(float t){
     t = t*0.95;
     return t;
}
```

Which seemed to work fine in most cases. A smaller factor times t (in our code, we use start temperature 50, for a small problem like $n = 5$) could make it so that promising solutions are reset because t would get quite small quickly. The function creates some randomness, but it solves quite some problems. Without this function, certain problems would get stuck in a certain state with no chance of getting out of there, because there is no other state with less conflicts, however sometimes you need to take a step back to take two forward. In conclusion, this is our code for simulated annealing (which is in the nqueensa.c file):

Listing 1: Simulated Annealing

```
int timeToTemperature(float t){
	t = t*0.95;
	return t;
}


void simulatedAnnealing(){
	float t = 50.0;
	for(int restart = 0; restart < 50; restart++){
		for(int tries = 0; tries < 500; tries++){
			for(int i = 0; i < nqueens; i++){
				int pos = columnOfQueen(i);
				int newPos = pos;
				// loop so that the random number is
				    not the same
				while (newPos == pos) {
					newPos = random() % nqueens;
				}
				// current = current amount of
				    conflicts
				int current = countConflicts();
				// next = amount of conflicts if
				    moved to random newPos
				moveQueen(i,newPos);
				int next = countConflicts();
				int deltaE = current - next;
				// if the amount of conflicts at the
				    new is bigger than
```

```
                // the old amount of conflicts, move the queen back to
                // the 'current' configuration
                if (deltaE < 0){
                    moveQueen(newPos,i);
                }
                // otherwise, the probability game starts, as
                // the next state has more conflicts than the
                // current one
                else{
                    float r = ((float)rand() / RAND_MAX);
                    float p = pow(euler, (deltaE/t));
                    if (p>r){
                        // do nothing, because the queen is already in
                        // newPos
                    }
                    else{
                        moveQueen(newPos,i);
                    }
                }
            }
            // if there are no conflicts, the solution is found
            if(!countConflicts()){
                break;
            }
            t = timeToTemperature(t);
        }
        if(!countConflicts()){
            break;
        }
        // retry with different start
        initiateQueens(1);
        t = 50.0;
    }
    if(!TESTING_MODE) printf("\nFinal State");
  printState();
}
```

3. The algorithm will find a solution often for a small nqueens problem, for example when n = 5 or 6. After this, it will struggle and find a solution every 5/6/7 tries or even more tries (or never) when the amount of queens

is big. For a large problem size, we choose a high starting temperature. Because it will take longer for the program to find a solution, the function timeToTemperature() is used more often, and so it will decrease more than you would maybe like to. For a small problem size a low starting temperature is better, as long as it is still good enough to make random decisions. This means that there may be less randomness, and there is at least a chance that you get out of a 'stuck' state.

4. The larger the amount of queens, the bigger the chance that there is some small mistake in one of the queen's placement, which influences the entire board. While the random 'hops' can solve these problems, this still only happens rarely, especially later on. And because the board is now at least 10x10, the chance that the 'perfect' random row and column is chosen is quite small.
   We tried a few things to make the program work better for larger problems. An idea to improve the program was to add an inner loop, which goes from $j = 0$ to $j = nqueens$. In this loop, we search for every column which has the least amount of conflicts, instead of using the randomized way that simulated annealing is based on. Another idea was to create an outer loop which breaks if it is solved, however we decided not to implement this. It could take a very long time before a solution is found.
   While it contributed, it did not work 'completely'. A problem with more than 10 queens will either take a long time or not find a solution at all. The full program is in Appendix B.

## 1.3 - Genetic Algorithm

### Considerations

There can be much variation in genetic algorithms. For instance, one could change the generation size, the maximum amount of generations or even whole algorithms involved. By trying more combinations, you could discover that for that specific problem, one modification might benefit results.
In our case, we noticed that generation size above a certain threshold does not seem to influence the result anymore. We noticed the same when we modified the maximum amount of generations we allowed. We also implemented two different mutation functions, which can be toggled using a macro.

### Design

We tried to make the algorithm as versatile as possible, so we can easily run tests. Therefore, we included macros that control the before named thresholds. Also, by adding command line arguments, we can easily run tests by changing little things.
For the algorithm itself, we used two mutation functions. One where every gene of a chromosome has a chance to be mutated, and one where there is a chance of mutating a single random gene. The last one makes mutation rarer, and

therefore less influential. This is benefitial when a generation of chromosomes is already close to convergence.
We also applied a little math to the randomness of the crossover function. We do generate each child as a crossover product of 2 parents, then apply mutation. However, when choosing the parents, we do not use the fitness function as a probability function. We apply a little math to give the 'good' chromosomes a much better chance to be selected than the 'bad' ones.
We did the following. We devide each chromosome's fitness (we will refer to this as $f$) by a value depending on its fitness ($y$). We want $y$ to be greater (say the size of the board, $y_0$) for lower values of $f$, and 1 for the maximum fitness ($f_m$). We can define a linear function

$$y = \frac{1 - y_0}{f_m} f + y_0$$

Since we know the maximum amount of the fitness is the same as the maximum amount of conflicts (see: `evaluateState()`), we can say that $f_m = \frac{y_0(y_0-1)}{2}$, and substitute

$$y = \frac{1 - y_0}{\frac{y_0(y_0-1)}{2}} f + y_0 = -\frac{2}{y_0} f + y_0$$

We can then find $p$ (probability), and simplify to speed up the calculation by the computer

$$p = \frac{f}{y} = \frac{f}{-\frac{2}{y_0} f + y_0} = \frac{f y_0}{-2f + y_0^2}$$

### Evaluation

For board sizes up to 10, the genetic algorithm solves the problem consistently. When going higher, it finds solutions which are often more than 90% correct. However, since it does not converge early because it found the solution, the time to terminate becomes exponentially larger, but remains managable up to a board size of about 50.
To conclude, we can say that this algorithm is best to find a solution that is close to correct. Meaning to say, that it does not always find the global optima, but tries to seek out and compare the best local optima. We conclude that the Genetic Algorithm works best for the N-queens problem.

## Exercise 2: Game of Nim

1. These graphs represent moves and evaluations: the nodes contain the state (value) and the evaluation and the edges are the moves. The nodes on the left represent the current player (the player that has to make a decision), where +1 is *MAX* and -1 *MIN*. A value of 1 is a terminal state and when a player is in such a state, the other play wins. *MAX* goes first. So for $n = 3$, *MAX* can either take 1 or 2 matches. Taking 2 matches results in a value of 1 and since it is then *MIN*'s turn, *MAX* wins and the evaluation

is +1. When *MAX* takes 1 match, *MIN* has only one option, that is take 1 match, and then *MAX* is in a terminal state, so *MIN* wins and the evaluation is -1. The maximum of -1 and +1 is +1, so, assuming both players play optimally, *MAX* wins.

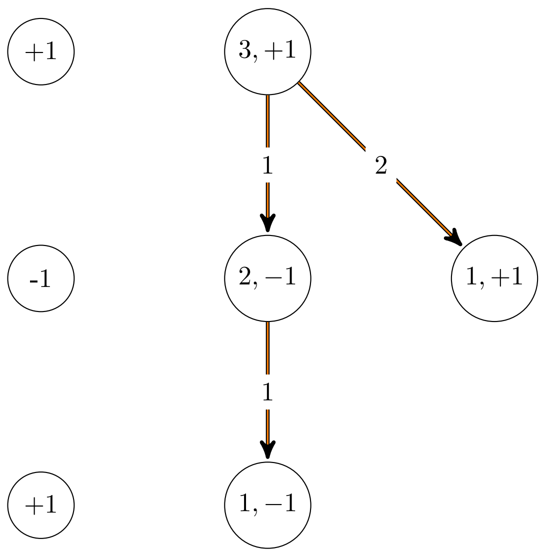



Figure 1: n=3

For $n = 4$, the three moves *MAX* can make result in values of 3, 2 and 1. Now *MIN* has to make a choice. 1 is terminal state, so in that case *MAX* wins and evaluate +1. In Figure 1 we see that if the value is 2 and it is *MIN*'s turn, the evaluation is -1. We can also see that if the value is 3 and it is *MAX*'s turn, the evaluation is +1, so if it is *MINS*'s turn, all the signs are inverted and so the evaluation is -1. The maximum of -1, -1 and +1 is +1, so *MAX* wins. The result is in Figure 2.

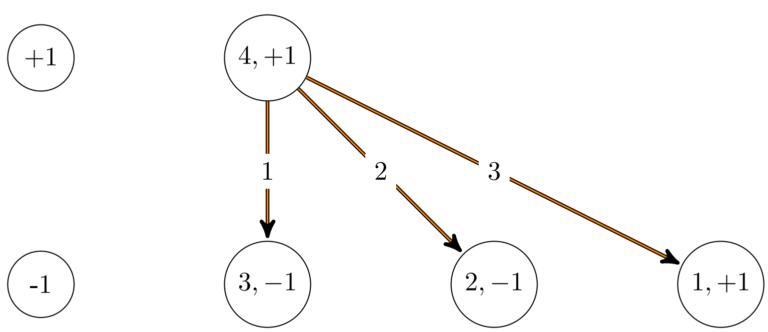



Figure 2: n=4

Using the same reasoning at $n = 4$, we can derive the graph for $n = 5$ (see Figure 3). The maximum of -1, -1 and -1 is -1, so *MIN* wins.

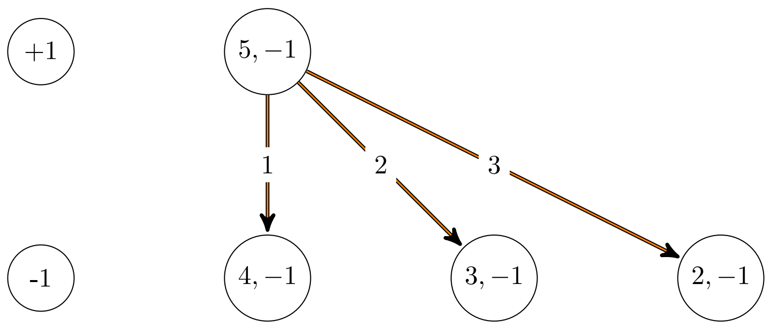



Figure 3: n=5

And again, using the same reasoning at $n = 4$ and $n = 5$, we can derive the graph for $n = 6$ (see Figure 4). The maximum of +1, -1 and -1 is +1, so *MAX* wins.

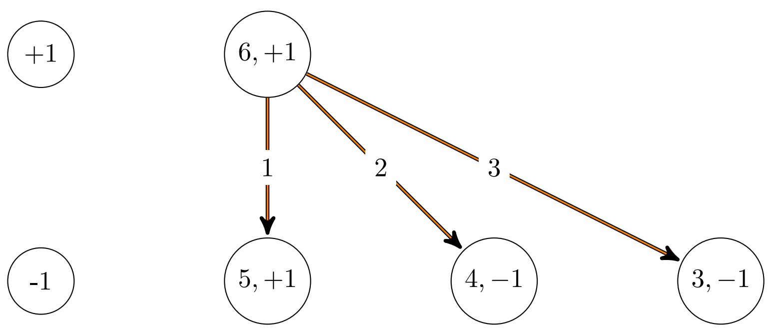



Figure 4: n=6

2. We can turn the *minimax* algorithm into the *negamax* algorothm by this rule: $min(a, b) = -max(-a, -b)$. The *negamax* function returns pairs (move + evaluation) and uses $colour \in \{-1, 1\}$, where -1 is *MIN* and 1 *MAX*. In our code, $pair[0]$ contains the move and $pair[1]$ the evaluation. This is a recursive function and our base case must return a pair: the evaluation is the opposite of the current colour, i.e. if 1 (*MAX*) is in the terminal state, -1 (*MIN*) wins and vice versa; the move isn't important in the base case, so we simply leave it as is (the best move and evaluation will be returned at the end of the function). Then, for every possible move, we check for the maximum or minimum evaluation. In this function, *minMax* is either a maximum or a minimum, depending on the colour. We don't want to assign the minimum/maximum of $pair[0]$ and *minMax* immediately to *minMax*, because we also want to determine the best move.

That's why we use an if-statement: $max(a, b) = a > b\,?\,a : b$. In the case that it's MAX' turn, we don't want to change this statement. If it's MIN's turn, we want to use: $min(a, b) = -max(-a, -b)$; and $-max(-a - b) = -a > -b\,?\,a : b$. $minMax$ has a starting value of either *-INFINITY* or *+INFINITY*, depending on the colour: if it's MAX' turn, $minMax$ is *-INFINITY* and *+INFINITY* at MIN's turn. We can acchieve this by assigning $-colour{*}INFINITY$ to $minMax$. The result of the function is in Listing 2. (see appendix C for full program)

Listing 2: NegaMax function

```c
int* negaMax (int state, int colour, int pair[2]) {
	int move, bestmove, minMax = -colour*INFINITY;
	/* terminal state ? */
	if (state == 1) {
		pair[1] = -colour; /* Max wins if min is in a terminal state */
		return pair;
	}
	/* non-terminal state */
	for (move = 1; move <= 3; move++) {
		if (state - move > 0) { /* legal move */
			pair = negaMax(state-move, -colour, pair);
			if (colour*pair[1] > colour*minMax) {
				minMax = pair[1];
				bestmove = move;
			}
		}
	}
	pair[0] = bestmove;
	pair[1] = minMax;
	return pair;
}

int minimaxDecisionNegaMax(int state, int turn) {
  int pair[2];
  if (turn == MAX) {
	negaMax(state, 1, pair);
    return pair[0];
  }
  /* turn == MIN */
  negaMax(state, -1, pair);
  return pair[0];
}
```

3. Results of running the program with n = 10,20,30,40,50:

```
n=10: quick          n=20: quick          n=30: a bit slower
10: Max takes 1      20: Max takes 3      30: Max takes 1
```

```
9: Min takes 1
8: Max takes 3
5: Min takes 1
4: Max takes 3
1: Min looses
```

```
17: Min takes 1
16: Max takes 3
13: Min takes 1
12: Max takes 3
 9: Min takes 1
 8: Max takes 3
 5: Min takes 1
 4: Max takes 3
 1: Min looses
```

```
29: Min takes 1
28: Max takes 3
25: Min takes 1
24: Max takes 3
21: Min takes 1
20: Max takes 3
17: Min takes 1
16: Max takes 3
13: Min takes 1
12: Max takes 3
 9: Min takes 1
 8: Max takes 3
 5: Min takes 1
 4: Max takes 3
 1: Min looses
```

n=40: The program is way too slow and no result is given (within a reasonable amount of time). n=50: There's no point in trying $n = 50$, since it will take an eternity to finish

Other things we observed: Min loses every game and Max takes a lot of 3s and Min a lot of 1s.

We now use a transposition table, which is an integer array of size 101 (index 0 not used), with another dimension of size 3, so that for every state we can store three values: best move, evaluation and colour. Now, in our $negaMax$ function, the first thing we do is check if have already evaluated this state. Initially all values of the transposition table are 0, so if the evaluation (which is either +1 or -1) is 0, that means that we haven't evaluated this state before. However, if it isn't 0, we already know what to do and return the move and evaluation. The move to make is the same for MIN and MAX, but the evaluation is different: in the transposition table, we also store for which player this evaluation is; if the current player is the same as the player in the transposition table, we return the evaluation as is, if it is the other player, however, we return the negation of the evaluation (if the evaluation at state $a$ is +1 for MAX, it will be -1 for MIN). After every possible move of this state has been checked, we can store the best move, the evaluation and the colour in the transposition table.

If we now run the program with $n = 50$ the program gives a result right away, so it definitely helps. The result of the function is in Listing 3. (see appendix D for full program)

Listing 3: NegaMax with transposition table

```
int* negaMax (int state, int colour, int pair[2], int
    transpositionTable[][3]) {
```

```
    if (transpositionTable[state][1] != 0) {
        pair[0] = transpositionTable[state][0];
        if (transpositionTable[state][2] == colour)
            pair[1] = transpositionTable[state][1];
        else
            pair[1] = -transpositionTable[state][1];
        return pair;
    }
    int move, bestmove, minMax = -colour*INFINITY;
    /* terminal state ? */
    if (state == 1) {
        pair[1] = -colour; /* Max wins if min is in a terminal state */
        return pair;
    }
    /* non-terminal state */
    for (move = 1; move <= 3; move++) {
        if (state - move > 0) { /* legal move */
            pair = negaMax(state-move, -colour, pair, transpositionTable);
            if (colour*pair[1] > colour*minMax) {
                minMax = pair[1];
                bestmove = move;
            }
        }
    }
    transpositionTable[state][0] = pair[0] = bestmove;
    transpositionTable[state][1] = pair[1] = minMax;
    transpositionTable[state][2] = colour;
    return pair;
}
```

# Appendix

## A: Exercise 1.1 output

```
Initial state:
......Q.
.....Q..
.....Q..
..Q.....
..Q.....
Q.......
..Q.....
..Q.....
```

```
Final State
.q......
...q....
.....q..
.......q
....Q...
Q.......
......Q.
..Q.....
```

```
Initial state:
.Q................................................
.............................Q....................
....................Q.............................
.............................................Q....
............................................Q.....
.................................................Q
..............Q...................................
.Q................................................
..........................Q.......................
........................................Q.........
.......Q..........................................
...............................q..................
......Q...........................................
...Q..............................................
.................................................Q
.........................Q........................
......................Q...........................
...................Q..............................
...............Q..................................
................................................q.
.............................Q....................
..........................................Q.......
.Q................................................
........Q.........................................
Q.................................................
............................................Q.....
..........Q.......................................
....................................Q.............
....................Q.............................
.........................................Q........
..........................................Q.......
........................Q.........................
......................Q...........................
...............Q..................................
...................Q..............................
................Q.................................
..............Q...................................
...................................Q..............
...................Q..............................
..........................................Q.......
..........................Q.......................
.............................Q....................
.......................Q..........................
................................Q.................
................................Q.................
.......................Q..........................
.........Q........................................
......Q...........................................
............................................Q.....
.........................Q........................
```

```
Final State
...........................q......................
......................................q...........
...........................................q......
....q.............................................
..................q...............................
.....q............................................
.............q....................................
.............................................q....
.................................q................
.................Q................................
..............................................q...
..q...............................................
..............................Q...................
............q.....................................
...............................................q..
............................Q.....................
...............................q..................
.....................q............................
........................................Q.........
................................................q.
..............q...................................
.....................................Q............
.q................................................
...Q..............................................
...........q......................................
........q.........................................
..........q.......................................
....................................q.............
..................................q...............
q.................................................
.........................................q........
........................q.........................
......................q...........................
...............q..................................
.......................................q..........
.......Q..........................................
.................................................Q
...................................q..............
...................q..............................
..........................................q.......
..........................Q.......................
.............................q....................
.........q........................................
................q.................................
......q...........................................
.......................q..........................
....................Q.............................
................................q.................
............................................q.....
.........................q........................
```

## B: full nqueens program (with hill climbing, simulated annealing and genetic algorithm)

```c
/* nqueens.c: (c) Arnold Meijster (a.meijster@rug.nl) */
```

```c

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

#define MAXQ 100

#define MAXGEN 5000
#define GENSIZE 50
#define MUTATE_MODE 1

#define TESTING_MODE 0

#define FALSE 0
#define TRUE 1

#define euler 2.7182818

#define ABS(a) ((a) < 0 ? (-(a)) : (a))

int nqueens;    /* number of queens: global variable */
int queens[MAXQ]; /* queen at (r,c) is represented by queens[r] == c */

void initializeRandomGenerator() {
  /* this routine initializes the random generator. You are not
   * supposed to understand this code. You can simply use it.
   */
  time_t t;
  srand((unsigned) time(&t));
}

/* Generate an initial position.
 * If flag == 0, then for each row, a queen is placed in the first column.
 * If flag == 1, then for each row, a queen is placed in a random column.
 */
void initiateQueens(int flag) {
  int q;
  for (q = 0; q < nqueens; q++) {
    queens[q] = (flag == 0 ? 0 : random() % nqueens);
  }
}

/* returns TRUE if position (row0,column0) is in
 * conflict with (row1,column1), otherwise FALSE.
 */
int inConflict(int row0, int column0, int row1, int column1) {
  if (row0 == row1) return TRUE; /* on same row, */
  if (column0 == column1) return TRUE; /* column, */
```

```c
  if (ABS(row0-row1) == ABS(column0-column1)) return TRUE;/* diagonal */
  return FALSE; /* no conflict */
}

/* returns TRUE if position (row,col) is in
 * conflict with any other queen on the board, otherwise FALSE.
 */
int inConflictWithAnotherQueen(int row, int col) {
  int queen;
  for (queen=0; queen < nqueens; queen++) {
    if (inConflict(row, col, queen, queens[queen])) {
      if ((row != queen) || (col != queens[queen])) return TRUE;
    }
  }
  return FALSE;
}

/* print configuration on screen */
void printState() {
  if(TESTING_MODE) return;
  int row, column;
  printf("\n");
  for(row = 0; row < nqueens; row++) {
    for(column = 0; column < nqueens; column++) {
      if (queens[row] != column) {
        printf (".");
      } else {
        if (inConflictWithAnotherQueen(row, column)) {
          printf("Q");
        } else {
          printf("q");
        }
      }
    }
    printf("\n");
  }
}

/* move queen on row q to specified column, i.e. to (q,column) */
void moveQueen(int queen, int column) {
  if ((queen < 0) || (queen >= nqueens)) {
    fprintf(stderr, "Error in moveQueen: queen=%d "
      "(should be 0<=queen<%d)...Abort.\n", queen, nqueens);
    exit(-1);
  }
  if ((column < 0) || (column >= nqueens)) {
    fprintf(stderr, "Error in moveQueen: column=%d "
      "(should be 0<=column<%d)...Abort.\n", column, nqueens);
    exit(-1);
  }
```

```c
  queens[queen] = column;
}

/* returns TRUE if queen can be moved to position
 * (queen,column). Note that this routine checks only that
 * the values of queen and column are valid! It does not test
 * conflicts!
 */
int canMoveTo(int queen, int column) {
  if ((queen < 0) || (queen >= nqueens)) {
    fprintf(stderr, "Error in canMoveTo: queen=%d "
      "(should be 0<=queen<%d)...Abort.\n", queen, nqueens);
    exit(-1);
  }
  if(column < 0 || column >= nqueens) return FALSE;
  if (queens[queen] == column) return FALSE; /* queen already there */
  return TRUE;
}

/* returns the column number of the specified queen */
int columnOfQueen(int queen) {
  if ((queen < 0) || (queen >= nqueens)) {
    fprintf(stderr, "Error in columnOfQueen: queen=%d "
      "(should be 0<=queen<%d)...Abort.\n", queen, nqueens);
    exit(-1);
  }
  return queens[queen];
}

/* returns the number of pairs of queens that are in conflict */
int countConflicts() {
  int cnt = 0;
  int queen, other;
  for (queen=0; queen < nqueens; queen++) {
    for (other=queen+1; other < nqueens; other++) {
      if (inConflict(queen, queens[queen], other, queens[other])) {
        cnt++;
      }
    }
  }
  return cnt;
}

/* evaluation function. The maximal number of queens in conflict
 * can be 1 + 2 + 3 + 4 + .. + (nquees-1)=(nqueens-1)*nqueens/2.
 * Since we want to do ascending local searches, the evaluation
 * function returns (nqueens-1)*nqueens/2 - countConflicts().
 */
int evaluateState() {
  return (nqueens-1)*nqueens/2 - countConflicts();
```

```c
}

/**********************************************************/

/* A very silly random search 'algorithm' */
#define MAXITER 1000
void randomSearch() {
  int queen, iter = 0;
  int optimum = (nqueens-1)*nqueens/2;

  while (evaluateState() != optimum) {
    printf("iteration %d: evaluation=%d\n", iter++, evaluateState());
    if (iter == MAXITER) break; /* give up */
    /* generate a (new) random state: for each queen do ...*/
    for (queen=0; queen < nqueens; queen++) {
      int pos, newpos;
      /* position (=column) of queen */
      pos = columnOfQueen(queen);
      /* change in random new location */
      newpos = pos;
      while (newpos == pos) {
        newpos = random() % nqueens;
      }
      moveQueen(queen, newpos);
    }
  }
  if (iter < MAXITER) {
    printf ("Solved puzzle. ");
  }
  printf ("Final state is");
  printState();
}

/**********************************************************/

int sharesColumnWithPreviousQueen(int row, int column){
        for(int i = 0; i < row; ++i){
                if(queens[i] == column) return TRUE;
        }
        return FALSE;
}

void hillClimbing() {
  int statelist[MAXQ];
  for(int i = 0; i < nqueens; ++i) {
          /* store state ranks in array statelist */
          for(int j = 0; j < nqueens; ++j) {
                  if(!sharesColumnWithPreviousQueen(i,j)) {
                          queens[i] = j;
                          statelist[j] = evaluateState();
```

```
                 } else {
                         statelist[j] = -1;
                 }
          }

          /* find amount of maximal ranks */
          int maxVal = -1, maxCnt = 0;
          for(int j = 0; j < nqueens; ++j) {
                  if(statelist[j] == maxVal) ++maxCnt;
                  if(statelist[j] > maxVal) {
                          maxVal = statelist[j];
                          maxCnt = 1;
                  }
          }

          /* determine a random position from the options
           * and move the queen there */
           if(maxCnt == 0) continue;
           int choice = random() % maxCnt;
           int cnt = 0;
           for(int j = 0; j < nqueens; ++j) {
                   if(statelist[j] == maxVal){
                           if(cnt == choice){
                                   queens[i] = j;
                                   break;
                           }
                           ++cnt;
                   }
           }
        }
        if(!TESTING_MODE) printf("\nFinal State");
        printState();
}

void randomRestartHillClimbing() {
    int maxRestarts = 25;
    int best[MAXQ];
    int bestRank = 0;
    for(int i = 0; i < maxRestarts; ++i) {
        int statelist[MAXQ];
        initiateQueens(1);
        for(int i = 0; i < nqueens; ++i) {
            /* store state ranks in array statelist */
            for(int j = 0; j < nqueens; ++j) {
              if(!sharesColumnWithPreviousQueen(i,j)) {
                  queens[i] = j;
                  statelist[j] = evaluateState();
              } else {
                  statelist[j] = -1;
              }
```

```
            }

            /* find amount of maximal ranks */
            int maxVal = -1, maxCnt = 0;
            for(int j = 0; j < nqueens; ++j) {
                if(statelist[j] == maxVal) ++maxCnt;
                if(statelist[j] > maxVal) {
                    maxVal = statelist[j];
                    maxCnt = 1;
                }
            }

            /* determine a random position from the options
            * and move the queen there */
            if(maxCnt == 0) continue;
            int choice = random() % maxCnt;
            int cnt = 0;
            for(int j = 0; j < nqueens; ++j) {
                if(statelist[j] == maxVal){
                    if(cnt == choice){
                        queens[i] = j;
                        break;
                    }
                    ++cnt;
                }
            }
        }
        int rank;
        if((rank = evaluateState()) > bestRank) {
            for(int i = 0; i < nqueens; ++i) {
                best[i] = queens[i];
            }
            bestRank = rank;
        }
    }
    for(int i = 0; i < nqueens; ++i) {
        queens[i] = best[i];
    }
    if(!TESTING_MODE) printf("\nFinal State");
    printState();
}

/*************************************************************/

int timeToTemperature(float t){
    t = t*0.95;
    return t;
}

void simulatedAnnealing(){
```

```
        for(int restart = 0; restart < 50; restart++){
                for(int tries = 0; tries < 1000; tries++){
                        for(int i = 0; i < nqueens; i++){
                                int current = countConflicts();
                                int min = current;
                                int minRow = i;
                                for (int j = 0; j < nqueens; j++){
                                        moveQueen(i,j);
                                        int currentRowConflict =
                                            countConflicts();
                                        if (currentRowConflict<min){
                                                min = currentRowConflict;
                                                minRow = j;
                                        }
                                }
                                moveQueen(i, minRow);
                        }
                        // if there are no conflicts, the solution is found
                        if(!countConflicts()){
                                break;
                        }
                }
                if(!countConflicts()){
                        break;
                }
                initiateQueens(1);
        }
        if(!TESTING_MODE) printf("\nFinal State");
    printState();
}

/*************************************************************/

void printChromosome(int* chrom){
    for(int i = 0; i < nqueens; ++i) {
        queens[i] = chrom[i];
    }
    printState();
}

void freeGeneration(int** chromosomes) {
    for(int i = 0; i < nqueens; ++i) {
        free(chromosomes[i]);
    }
    free(chromosomes);
}

void initializeGeneration(int** generation) {
    for(int i = 0; i < GENSIZE; ++i) {
        for(int j = 0; j < nqueens; ++j) {
```

```
            generation[i][j] = random() % nqueens;
        }
    }
}

int calcFitness(int* chrom) {
    for(int i = 0; i < nqueens; ++i) {
        queens[i] = chrom[i];
    }
    return evaluateState();
}

int getFittestIndex(int** chromosomes) {
    int idx = 0, fitness = calcFitness(chromosomes[0]);
    for(int i = 1; i < GENSIZE; ++i) {
        int curFitness = calcFitness(chromosomes[i]);
        if(curFitness > fitness) {
            idx = i;
            fitness = curFitness;
        }
    }
    return idx;
}

/* Mutates one gene in 1/4 new chromosomes OR
 * Mutates a gene with a probability of 10%
 */
void mutate(int* chrom){
    if(!MUTATE_MODE) {
        if(!(random() % 4)) {
            chrom[random() % nqueens] = random() % nqueens;
        }
    } else {
        for(int i = 0; i < nqueens; ++i) {
            if(!(random() % 10)) {
                chrom[i] = random() % nqueens;
            }
        }
    }
}

int* crossover(int** a, int** b){
    int cutPoint = random() % nqueens;
    int* new = malloc(nqueens * sizeof(int));
    for(int i = 0; i < nqueens; ++i) {
        if(i < cutPoint) {
            new[i] = (*a)[i];
        } else {
            new[i] = (*b)[i];
        }
```

```
    }
    return new;
}

/* returns a new generation of chromosomes, and frees the given one */
int** reproduce(int** chromosomes, int generationIdx){
    int** newGeneration = malloc(GENSIZE * sizeof(int*));
    for(int i = 0; i < GENSIZE; ++i) {
        newGeneration[i] = malloc(nqueens * sizeof(int));
    }

    /* instead of using the fitness as a probability function,
     * we will use function of the fitness and some other variables
     * to give better chromosomes a greater advantage over the
     * worse chromosomes. This is to decrease the time of convergence */
    int* probabilities = malloc(GENSIZE * sizeof(int));
    int totalProb = 0;
    for(int i = 0; i < GENSIZE; ++i) {
        /* probability function yields values between [nqueens, 1]
         * for lowest and highest fitne sses respectively */
        int f = calcFitness(chromosomes[i]);
        probabilities[i] = (f * nqueens) / (-2*f + nqueens*nqueens);
        totalProb += probabilities[i];
    }

    for(int i = 0; i < GENSIZE; ++i) {
        int cr1 = random() % totalProb;
        int cr2 = random() % totalProb;
        int a = 0, b = 0;
        for(int j = 0; j < GENSIZE; ++j) {
            if(cr1 < probabilities[j] && cr1 >= 0) a = j;
            if(cr2 < probabilities[j] && cr2 >= 0) b = j;
            cr1 -= probabilities[j];
            cr2 -= probabilities[j];
        }
        // preventing inbreeding
        if(a == b) {
            --i; continue;
        }
        newGeneration[i] = crossover(&chromosomes[a], &chromosomes[b]);
        mutate(chromosomes[i]);
    }

    free(probabilities);
    freeGeneration(chromosomes);
    return newGeneration;
}

void geneticPermutation(){
    int** chromosomes = malloc(GENSIZE * sizeof(int*));
```

```
    for(int i = 0; i < GENSIZE; ++i) {
        chromosomes[i] = malloc(nqueens * sizeof(int));
    }
    initializeGeneration(chromosomes);

    int maxFitness = (nqueens-1)*nqueens/2;
    for(int i = 0; i < MAXGEN; ++i) {
        chromosomes = reproduce(chromosomes, i);
        int max = getFittestIndex(chromosomes);
        if(calcFitness(chromosomes[max]) == maxFitness) {
            printf("\nFound solution");
            printChromosome(chromosomes[max]);
            return;
        }
    }
    int fittest = getFittestIndex(chromosomes);
    printf("\nTerminated with %d/%d (%.2f%%)",
         calcFitness(chromosomes[fittest]), maxFitness,
         100*calcFitness(chromosomes[fittest]) / (float)maxFitness);
    printChromosome(chromosomes[fittest]);

    freeGeneration(chromosomes);
}


int main(int argc, char *argv[]) {
  int algorithm;

  do {
    if(!TESTING_MODE) printf ("Number of queens (1<=nqueens<%d): ",
         MAXQ);
    if(argc == 3)
        nqueens = atoi(argv[1]);
    else
        scanf ("%d", &nqueens);
  } while ((nqueens < 1) || (nqueens > MAXQ));

  do {
    if(!TESTING_MODE) {
        printf ("Algorithm: (1) Random search (2) Hill climbing ");
        printf ("(3) Simulated Annealing (4) Genetic Permutation: ");
    }
    if(argc == 3)
        algorithm = atoi(argv[2]);
    else
        scanf ("%d", &algorithm);
  } while ((algorithm < 1) || (algorithm > 4));

  initializeRandomGenerator();

```

```c
  initiateQueens(1);

  if(!TESTING_MODE) printf("\nInitial state:");
  printState();

  switch (algorithm) {
  case 1: randomSearch();           break;
  case 2: randomRestartHillClimbing(); break;
  case 3: simulatedAnnealing();     break;
  case 4: geneticPermutation();     break;
  }

  if(TESTING_MODE) {
    FILE *out = fopen("1.out", "a");
    fprintf(out, "{%.1f,%d}", (double)nqueens/10, countConflicts());
    fclose(out);
  }
  return 0;
}
```

### C: Nim with negaMax function [nimNegaMax.c]

```c
#include <stdio.h>
#include <stdlib.h>

#define MAX 0
#define MIN 1

#define INFINITY 9999999

int* negaMax (int state, int colour, int pair[2]) {
	int move, bestmove, minMax = -colour*INFINITY;
	/* terminal state ? */
	if (state == 1) {
		pair[1] = -colour; /* Max wins if min is in a terminal
		     state */
		return pair;
	}
	/* non-terminal state */
	for (move = 1; move <= 3; move++) {
		if (state - move > 0) { /* legal move */
			pair = negaMax(state-move, -colour, pair);
			if (colour*pair[1] > colour*minMax) {
				minMax = pair[1];
				bestmove = move;
			}
		}
	}
	pair[0] = bestmove;
```

```
        pair[1] = minMax;
        return pair;
}

int minimaxDecisionNegaMax(int state, int turn) {
  int pair[2];
  if (turn == MAX) {
        negaMax(state, 1, pair);
    return pair[0];
  }
  /* turn == MIN */
  negaMax(state, -1, pair);
  return pair[0];
}

void playNim(int state) {
  int turn = 0;
  while (state != 1) {
    int action = minimaxDecisionNegaMax(state, turn);
    printf("%d: %s takes %d\n", state,
           (turn==MAX ? "Max" : "Min"), action);
    state = state - action;
    turn = 1 - turn;
  }
  printf("1: %s looses\n", (turn==MAX ? "Max" : "Min"));
}

int main(int argc, char *argv[]) {
  if ((argc != 2) || (atoi(argv[1]) < 3)) {
    fprintf(stderr, "Usage: %s <number of sticks>, where ", argv[0]);
    fprintf(stderr, "<number of sticks> must be at least 3!\n");
    return -1;
  }

  playNim(atoi(argv[1]));

  return 0;
}
```

### D: Nim with transposition table [nimTrans.c]

```
#include <stdio.h>
#include <stdlib.h>

#define MAX 0
#define MIN 1

#define INFINITY 9999999

```

```
int* negaMax (int state, int colour, int pair[2], int
    transpositionTable[][3]) {
        if (transpositionTable[state][1] != 0) {
                pair[0] = transpositionTable[state][0];
                if (transpositionTable[state][2] == colour)
                        pair[1] = transpositionTable[state][1];
                else
                        pair[1] = -transpositionTable[state][1];
                return pair;
        }
        int move, bestmove, minMax = -colour*INFINITY;
        /* terminal state ? */
        if (state == 1) {
                pair[1] = -colour; /* Max wins if min is in a terminal
                    state */
                return pair;
        }
        /* non-terminal state */
        for (move = 1; move <= 3; move++) {
                if (state - move > 0) { /* legal move */
                        pair = negaMax(state-move, -colour, pair,
                            transpositionTable);
                        if (colour*pair[1] > colour*minMax) {
                                minMax = pair[1];
                                bestmove = move;
                        }
                }
        }
        transpositionTable[state][0] = pair[0] = bestmove;
        transpositionTable[state][1] = pair[1] = minMax;
        transpositionTable[state][2] = colour;
        return pair;
}

int minimaxDecisionNegaMax(int state, int turn, int
    transpositionTable[][3]) {
  int pair[2];
  if (turn == MAX) {
        negaMax(state, 1, pair, transpositionTable);
    return pair[0];
  }
  /* turn == MIN */
  negaMax(state, -1, pair, transpositionTable);
  return pair[0];
}

void playNim(int state, int transpositionTable[][3]) {
  int turn = 0;
  while (state != 1) {
    int action = minimaxDecisionNegaMax(state, turn, transpositionTable);
```

```c
    printf("%d: %s takes %d\n", state,
          (turn==MAX ? "Max" : "Min"), action);
    state = state - action;
    turn = 1 - turn;
  }
  printf("1: %s looses\n", (turn==MAX ? "Max" : "Min"));
}

int main(int argc, char *argv[]) {
  if ((argc != 2) || (atoi(argv[1]) < 3)) {
    fprintf(stderr, "Usage: %s <number of sticks>, where ", argv[0]);
    fprintf(stderr, "<number of sticks> must be at least 3!\n");
    return -1;
  }

  int transpositionTable[101][3] = {{0}};

  playNim(atoi(argv[1]), transpositionTable);

  return 0;
}
```