Artificial Intelligence 1
Lab 2

Roeland Lindhout (s2954524) & Younes Moustaghfir (s2909758)
AI2/AI3

May 16, 2016

N-queens problem

The problem at hand is about the n-queens problem. It is an expansion of the 8-queens problem, which has often been discussed in previous courses. The goal is to place "n" number of queens on an n*n sized board, in such a way that they are not able to hit each other. For sizes bigger than 4, more solutions are possible.

The way to solve such a problem can be done in various ways. One approach, is to randomly place queens on the board until they accidentally have one of the right configurations. Obviously, this can be done more effectively. The first way we try to solve this, is by the hill-climbing approach. This basically means, that for every row, we look at which position is better than the previous position, and then choose that one. Still a blunt way to solve the problem, but already more sophisticated than randomly trying. The problem with this, is that it often gets stuck in local maxima. The second approach, is by simulated annealing. Simulated annealing is similar to hill-climbing, but it includes random moves, to avoid the problem with the local maxima. The way you decide when to make a random jump, is based on a relationship between the "temperature" (in this case, the number of correct queens) and the elapsed time. As the time increases, less random jumps are made. The final approach is with a generic algorithm. A genetic algorithm works much more differently. First of all, it does not use an initial state, but an initial population. This initial population is then sorted on their "fitness", which is the number of correctly placed queens. The best few are allowed to reproduce, i.e. make new states. This is done by combining two states. There is also a small chance of a random mutation in the newly produced state. By mixing up the parents and including the mutations, the population will approach the solution state, until one of the children is the correct solution.

0.1 Hill Climbing algorithm

We will be solving this problem using three different algorithms. First, we will use Hill Climbing to solve this. This is a quite straightforward algorithm. If, in your search space, the neighbour has a higher value, go to that neighbour and repeat that process until you've found a maximum. In the case of the n-queens problem, you look at all the positions in a column and select the one with the highest value.

For this algorithm, we look at each neighbouring state of the current column. We then evaluate each position that the queen can go to. We do that for each queen and then move that queen to the position with the highest evaluation.

This approach is quite simple and therefore not that effective. The algorithm will not always find a solution. For the unmodified code, 6 out of 20 times in average, the algorithm was able to find a solution. We've therefore tried some improvements for the code. Hill Climbing can get stuck in local optima and then it's unable to find the global optimum. We therefore implemented something that when the algorithm is stuck, it chooses a random queen to randomly move so that is might get out of this local optimum. With this improvement, we see its success rate in Table 1. This table is in the Appendix

This algorithm does quite well up to 7 queens, after that it's performance declines. The modified code did better than the original code.

0.2 Simulated Annealing algorithm

Simulated Annealing is somewhat the same as Hill Climbing, but better. Hill Climbing is able to get stuck in local maxima, therefore not always returning the optimal solution. Simulated Annealing allows "bad moves". With these "bad moves", it is able to escape local maxima and find a global maximum.

For Simulated Annealing, we figured that the temperature function should be something like: $T(t) = T0 * t$, where we filled in 400 for the starting temperature and t should be something around 0.99. We have used this function to lower the temperature until it has reached a certain value, in our case: 0.001. This sets the maximum amount of iterations, because if the temperature drops below that certain value, the while loop of our program stops. With these values, SA is quite similar to Hill Climbing. The difference is that we have implemented a function $ExpMove()$ that decides whether a "bad move" should be made. This is where we differ from the pseudocode as well. We look at a current position and the maximum of a column to decide whether a bad move is accepted.

For simulated Annealing, we have noticed a few things. First of all, the algorithm usually finds a solution within 50 steps. When it did not find a solution after 50 steps, its chance of finding one is very small. As might have been clear by the previous sentence, the program does not always find a solution. For a number of queens above 10 this only gets worse. Why does it get worse? For more than 10 queens, the search space gets too big for our algorithm and the "bad move" that the algorithm takes will help the algorithm.

Genetic algorithm

The genetic algorithm consists of three functions. The first function describes the mutation. The mutation is as random as possible. It takes a random queen, and places this at a random, new position, similar to how the random-search algorithm was implemented. Next, there is a function to sort the population. The population is sorted based on the evaluateState function from the given program. This evaluation is stored on the last place in the array. For every item in the array, the evaluation is compared to the evaluation of the next one, and then shifted until they are all in the right place. The next function is the real implementation of the program. First, a 2D array is made, with a size of the number of queens to the power of 2. This is chosen, because the number of different places the queens can stand, also increases exponentially. The first dimension represents the population, the second represents the configuration of the separate parents. The parents are initially randomly produced. After this, the population is sorted and the crossover can begin. The crossover is done with the best 20 % of the population. This was chosen to keep a fairly large amount of different configurations, but not too much. The crossover is done in pairs, so the first of the population pairs with the second, the third with the fourth and so forth. A random number between 0 and the number of queens is chosen. The first parent gives it's rows up to and including this number, and the second parent starting from this number. In this way, the child is a crossover between the parents. The child is placed at the last place in the population, thereby replacing the worst of the population. Then, 4 % of the children receive a mutation. This seems like a high number, but we thought it was necessary, because the initial population is randomly produced, and therefore it is quite possible that there are position that are not included in any of the parents, and we try to include these positions in the population by these mutations. Then, the population is once again sorted, and the process continues until the solution is found. The population is immediately sorted after the child is made, so the process is sped up, if the child is better than the first 20 %. Finally, the final state is printed.

The program is very fast for problems up to 8. It is also able to solve problems with 8 and 9 queens, but this can take quite long. It does always find a solution. It all depends on the random initialization of the population. If this is done very unfortunately, it takes longer. Problems bigger than 9, have not found a solution yet. We tried a problem with 10 queens, but after 15 minutes we gave up, expecting it not to find a solution ever. The valgrind output is the following for n = 8 :

```
Number of queens (1<=nqueens<100): 8
Algorithm: (1) Random search (2) Hill climbing (3)
    Simulated Annealing (4) Genetic algorithm: 4
Final state is
....q...
......q.
.q......
...q....
.......q
q.......
..q.....
```

```
.....q..
==26906==
==26906== HEAP SUMMARY:
==26906==     in use at exit: 0 bytes in 0 blocks
==26906==   total heap usage: 65 allocs, 65 frees, 2,816
    bytes allocated
==26906==
==26906== All heap blocks were freed -- no leaks are
    possible
==26906==
==26906== For counts of detected and suppressed errors,
    rerun with: -v
==26906== ERROR SUMMARY: 0 errors from 0 contexts
    (suppressed: 0 from 0)
```

Nim

The game of Nim is about a stack of items, from which each player can take 1 to 3 items each turn, after which the other player's turn is. You lose when you have to take the last item. The players in our program are thought to be playing optimally. Playing optimally means that in each situation, the best possible choice is made, based on the best possible choice in the next turn of the opponent. Doing this, it can quickly be predicted who wins. For n=3, MAX, who begins, will take 2, leaving only 1 for MIN, who then loses. For n=4, MAX will take 3, once again leaving only one for MIN, who will once again lose. For n=5, assuming optimal play, MIN will win. First, MAX will take 3, and then MIN will take 1, leaving only 1 for MAX to take. No matter what MAX does at his/her first turn, MIN will be able to do a finishing move. For n=6, MAX will win. MAX will start by taking 1 item, which will result in the same situation as discussed before this one, only from the perspective of MIN.

Program description and evaluation

The program is fairly simple. We kept the basic structure of the given program, but we placed everything into one function. First, it is checked if the state is one. In this case, MAX has lost, and therefore -1 is returned. If this is not the case, the different moves are evaluated. Each move is recursively passed onto the function again, only negated, to show that it is MIN's turn. This is assigned to a variable. If this is variable is better than best, this variable becomes the new best. Initially, best was set to -∞. The move that was the best, is stored in the variable bestmove. This is printed at the end. At this point, all moves have been decided, and printed at once. This function is used the same way as in the original program, with the only change that it does not use the variable turn.

The program always find a solution, but somehow not always the most optimal solution, from MAX's point of view. The more items there are at the beginning, the longer it takes to find an answer. Up to ± 35, the time it takes is not too long, but after this, the program becomes very slow.

Source code

Listing 1: nqueens.c

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

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

#define MAXQ 100

#define FALSE 0
#define TRUE 1

#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.
```

```c
 */
int inConflict(int row0, int column0, int row1, int column1) {
  if (row0 == row1) return TRUE; /* on same row, */
  if (column0 == column1) return TRUE; /* column, */
  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() {
  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);
  }
  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();
}

int selectRandom(int n) {
  int i;
  i = 0 + random() % (n-0);
  return i;
}

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

/* 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();
}

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

void hillClimbing() {
  int newqueen, newpos, pos, ev;
  int queen, iter = 0;
  int optimum = (nqueens-1)*nqueens/2;
  int max = 0;
  int i ,x;
```

```

  while ((evaluateState()) != optimum) {
    printf("iteration %d: evaluation=%d\n", iter++, evaluateState());
    if (iter == MAXITER) break; /* give up */
    ev = evaluateState();
    for (queen=0; queen < nqueens; queen++) {
      pos = columnOfQueen(queen);
      for(i = 0; i < nqueens; i++) {
        moveQueen(queen, i);
        if(evaluateState() > max) {
          newpos = i;
          max = evaluateState();
          newqueen = queen;
        }
        else if (evaluateState() == max) {
          x = random() % 2;
          switch (x) {
            case 0:
              newpos = i;
              break;
            case 1:
              newpos = random() % nqueens;
              break;
          }
          newqueen = queen;
        }
      moveQueen(queen, pos);
      }
      if (evaluateState() == ev) {
        moveQueen(queen, random() %nqueens);
    }

    }
    moveQueen(newqueen,newpos);
  }
  if (iter < MAXITER) {
    printf ("Solved puzzle. ");
  }
  printf ("Final state is");
  printState();
}


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

int ExpMove(int dE, double iter) {
  int random1;
  double E;
  E = exp((dE/iter)/nqueens*nqueens) * 100;
  random1 = random() % 101;
```

```
  if(E > random1) {
    return 1;
  }
  else {
    return 0;
  }
}

void simulatedAnnealing() {
  int dE, newqueen, ev;
  int queen, iter = 0, i;
  int optimum = (nqueens-1)*nqueens/2;
  int max = 0, current;
  double temp = 50.0, alpha = 0.95;
  double epsilon = 0.01;

  while (temp > epsilon) {
    ev = evaluateState();
    printf("iteration %d: evaluation=%d\n", iter++, evaluateState());
    if(ev == optimum) break;
    int newpos;
    for (queen=0; queen < nqueens; queen++) {
      int pos;
      /* position (=column) of queen */
      pos = columnOfQueen(queen);
      for(i = 0; i < nqueens; i++) {
        moveQueen(queen, i);
        current = evaluateState();
        if(current > max) {
          newpos = i;
          max = evaluateState();
          newqueen = queen;
        }
        dE = max - current;
        if(ExpMove(dE, temp)) {
          newpos = random() % nqueens;
        }
        if(dE < 0) {
          if(ExpMove(dE, temp)) {
            newpos = random() % nqueens;
          }
        }
        moveQueen(queen, pos);
      }

    }
    moveQueen(newqueen, newpos);
    temp *= alpha;
  }
  if (ev == optimum) {
```

```
    printf ("Solved puzzle. ");
  }
  printf ("Final state is");
  printState();
}


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

void mutation () {
  // a random queen is moved to a random, new position (based on randomSearch)
  int pos,newpos,queen;
  queen = random() % nqueens;
  pos = columnOfQueen(queen);
  newpos = pos;
  while (newpos == pos) {
    newpos = random() % nqueens;
  }
  moveQueen(queen, newpos);
}

void sortPopulation (int size, int **arr) {
  int i, n, value;
 // Population is sorted on the evaluated stated, which is stored in the last position of the array.
  for (i = 1; i < size; i++) {
    value = arr[i][nqueens];
    n = i;
    while ((n > 0) && (arr[n-1][nqueens] > value)) {
      arr[n] = arr[n-1];
      n--;
    }
    arr[n] = arr[i];
  }
}


void geneticAlgorithm() {

  int optimum = (nqueens-1)*nqueens/2;
  int m;
  int i, n, q;
  int **arr;
  int size = pow(nqueens,2);

  arr = malloc(size*sizeof(int *));
  assert(arr != NULL);
  // make initial population of size 100
  for(i = 0; i < size; i++) {
```

```c
        arr[i] = malloc((nqueens+1)*sizeof(int));
        assert(arr[i] != NULL);
    for (q = 0; q < nqueens; q++) {
      arr[i][q] = random() %nqueens;
    }
    arr [i][nqueens] = evaluateState();
    }

    sortPopulation(size, arr);

/* Cross-over:pick a random queen n, then the positions of the queens after n of 1 parent
and in front of n of the other parent */

  while (evaluateState() != optimum ) {
    // The best 20% of the population can reproduce
    for (i = 0; i < size/5; i+=2) {
      int randomPlace = random () %nqueens;
      for (n = 0; n <= randomPlace; n++) {
        // the worst population members are hereby deleted
        arr [size-1][n] = arr[i][n];
        arr [size-1][nqueens-n] = arr[i+1][nqueens-n];
        // random mutation occurs 4% of the time
        m = random() % 100;
        if (m < 5) {
          mutation();
        }
        arr[size-1][nqueens] = evaluateState();
        sortPopulation(size, arr);
      }
    }
  }

  printf ("Final state is");
  printState();

  for (i = 0; i < size; i++) {
    free(arr[i]);
  }
  free(arr);
}

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


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

  do {
    printf ("Number of queens (1<=nqueens<%d): ", MAXQ);
```

```
    scanf ("%d", &nqueens);
  } while ((nqueens < 1) || (nqueens > MAXQ));

  do {
    printf ("Algorithm: (1) Random search (2) Hill climbing ");
    printf ("(3) Simulated Annealing (4) Genetic algorithm: ");
    scanf ("%d", &algorithm);
  } while ((algorithm < 1) || (algorithm > 4));

  initializeRandomGenerator();


  if (algorithm != 4) {
    initiateQueens(1);
    printf("\nInitial state:");
    printState();
  }

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

  return 0;
}
```

Listing 2: nim.c

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

#define MAX 0
#define MIN 1

#define INFINITY 9999999

int negaMax(int state) {
  int move, bestmove, best = -INFINITY;
  int m;
  if (state == 1) {
    return -1;
  }
  for (move = 1; move <= 3; move++) {
    if (state - move > 0) {
      m = -negaMax(state - move);
      if (m > best) {
        best = m;
```

```
        bestmove = move;
      }
    }
  }
  return bestmove;
}


void playNim(int state) {
  int turn = 0;
  while (state != 1) {
    int action = negaMax(state);
    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;
}
```

Appendix

Table 1: Succes rate of modified algorithm

Number of Queens	Succesrate
4	100%
5	100%
6	100%
7	100%
8	50%
9	35%
10	20%
11	20%
12	10%
13	5%