Artificial Intelligence 1
Lab 2

Eduardo Bier s3065979 & Auke Roorda s2973782
CS1

May 15, 2016

Theory

Hill Climbing

1. The algorithm is unable to solve the problem most of the time, even for a small number of queens. As shown by Table 1, the algorithm's ability to solve the problem quickly drops as the number of queens rises, reaching a point where it is extremely unlikely to solve it.

number of queens	solved puzzles
4	26.5%
6	4.4%
8	3.7%
11	0.6%
12	0.4%
16	0.0%
32	0.0%

Table 1: Percentage of solved puzzles vs number of queens.

2. The algorithm does not solve the problem when it chooses a wrong position for one of the queens. The number of wrong positions clearly rises as the number of queens rises, so it makes mistakes more often. In other words, the algorithm ends up getting stuck on local maximums and never reaches the global maximum. To fix this problem, if the algorithm does not find a solution it resets the board and tries again until a certain (arbitrary) limit is reached. This seems to work rather well, even for a small limit, as shown in Table 2. However, for a large numbers of queens, the algorithm still struggles to find a solution.

3. Table 2 shows the results after implementing the repetitions in the algorithm.

number of queens	solved puzzles
4	86.7%
6	29.0%
8	27.3%
11	4.7%
12	3.6%
16	0.9%
32	0.0%

Table 2: Percentage of solved puzzles vs number of queens using a limit of 7 repetitions.

Simulated Annealing

1. The simulated annealing algorithm was a pretty straight-forward adaptation of the pseudo-code. In order to pick a successor, all the algorithm does is pick a random queen and a new random position for it. In order not to be stuck in the same position, the new position of that queen is forced to be different from its current position. The pseudo-code was also modified so as to always move the queen and undo the move if it's not a good move and with probability $1 - e^{\frac{\Delta E}{T}}$. The reason for that was how the evaluateState() function worked, the state needed to be change in order to be evaluated, so it was needed to determine ΔE.

2. The function T(t) is responsible for how many times the algorithm does a bad move. If the temperature lowers too fast, then the algorithm might end up in one of the local maximums. On the other hand, if the temperature falls too slowly, then the algorithm does a lot of bad moves and ends up too far away from a solution. Ideally, the temperature should drop fast on the beginning and slow down later on. For that reason, a linear temperature function is not ideal for the simulated annealing.
   In order to make the timeToTemperature() independent of the function implemented by T(t), this method loops through time until it reaches the desired temperature (or lower).

3. The algorithm works most of the time for small numbers of queens and a exponential function, as shown in Table 3. Since the algorithm works well for small ($<= 16$) number of queens, even with low starting temperatures, those should be used to do run the algorithm. However, when the number of queens is big, higher starting temperatures are preferable, since one can get more than double the chance of success. Note that higher starting temperatures means more time taken to run the algorithm.

4. The program doesn't work well for larger problem sizes because a high number of queens means that picking the right queen to move to a good position is less likely to happen.

starting temperature	number of queens	solved puzzles
10	4	100.0%
10	8	91.5%
10	16	44.9%
10	32	1.7%
10000	4	100.0%
10000	8	92.0%
10000	16	40.9%
10000	32	2.2%
10000000	4	100.0%
10000000	8	92.3%
10000000	16	45.3%
10000000	32	3.6%

Table 3: Percentage of solved puzzles vs number of queens with varying starting temperatures.

Genetic Algorithm

1. The genetic algorithm used in this program was implemented with a population size of 1000 after testing a few other possibilities. A larger population would mean longer time to process run it, while a smaller size would often give worse results. The algorithm uses all forms of mutation in a random way so as to be able to create a wide range of different offsprings. The rate of such mutations ($\frac{1}{\text{population size}}$) was also determined empirically and seems to work well, since it usually produces at least 1 mutated child. This allows the algorithm to escape local maximums without completely throwing it off track. In order to choose which individuals to reproduce, the algorithm tends to picks them randomly, but fitter individuals have better probabilities of being chosen. The chance of a certain individual to be picked is given by $\frac{\text{fitness}}{\text{total fitness}}$.
   The genetic algorithm works extremely well for a low number of queens, being able to solve the problem with a very high rate of success with little computation time, as shown in Table 4. However, the algorithm scales horribly with higher complexity, specially if the limit of iterations depends on the number of queens. Not only does it take a very long time to run, but it also rarely finds a solution. For that reason, the simulated annealing ends up being a much better option, since it provides much better success rate and does not take so long to run.

Game of Nim

a) Assuming optimal play, Max will win the games starting with 3, 4 and 6 matches, while Min will win the one starting with 5 matches. For the first two games, starting with 3 and 4 matches, Max can win by taking 2 and 3 matches respectively. For the third game, however, there is no way for Max

number of queens	solved puzzles	time(1)
4	100%	2.999s
6	99%	8.703s
8	100%	18.329s
11	49%	2m17.266s
12	45%	2m13.136s
16	6%	3m8.636s
32	0%	3m40.740s

Table 4: Percentage of solved puzzles vs number of queens using the genetic algorithm. (1) Time needed to run the algorithm 100 times using a maximum of 100 iterations per run.

to win, since whatever move he makes Min will receive a winning hand: by taking 1 match, Min can take 3 later and win; by taking 2 matches, Min can take 2 matches and win; and by taking 3 matches, Min can take 1 match and win. Finally, for a game starting with 6 matches, Max can win by taking 1 match and Min will receive a losing hand (with 5 matches), as shown before.

c) The algorithm takes a very long time to process the games starting with 40 and 50 matches because it ends up simulating the game for the same states over and over again. As so, a transposition table, a table where the return value of analyzed states are kept, really speeds up the process. After enhancing the algorithm with a transposition table it is definitely much faster, giving almost instant results, even for numbers much larger than 50.

Programming

N Queens Problem

Program description

The N Queens problem consists of positioning queens on a N x N board in such a way that none of them collide (according to the chess rules). In order to solve this problem a few different methods were implemented, allowing for an easy way to compare them.

Problem analysis

Finding the correct positions of the n queens is no easy task. In fact, even for small numbers such as 6 and 7, the problem is non-trivial. The methods implemented in the program, however, can tackle this problem for a large number of queens. The program implements 4 algorithms to do so: random search, hill climbing, simulated annealing and a genetic algorithm.

Program design

The user is able to pick which method is used to try to find a solution. Aside from that, the user is also prompted to determine the number of queens used. Because the algorithms have a chance of failing to find a solution, the program also supports an optional argument when the program is executed to determine the number of times it will run the desired algorithm. This feature also proved to be useful when comparing the different algorithms.
The hill climbing part of the algorithm was designed with repetition in mind, since it often did not find a solution, specially for large values of n. By resetting the board and trying again, the algorithm was able to greatly improve it's success rate.
For the simulated annealing, a lot of schedule functions were tested until a decision was made to lower the temperature by 5% per iteration. This exponential function has the advantages discussed in the theory part of this report and, as such, had a much better performance than other functions.
In order to implement the genetic algorithm a lot of new functions had to be created. Not only were the select, mutate and reproduce functions, which are essential to the genetic algorithm, needed, but also functions related to the sorting of a population. A new type was also created to make such functions easier to implement.

Program evaluation

The program works rather well for values of $n <= 32$, depending on the algorithm chosen. However, for bigger values it seems as though the program rarely finds a solution.

Program output

Listing 1: Output for n = 4 using Hill Climbing

```
Number of queens (1<=nqueens<100): 4
Algorithm: (1) Random search (2) Hill climbing (3) Simulated Annealing
    (4) Genetic Algorithm: 2

Initial state:
.q..
...Q
q...
...Q
Puzzle not solved. Final state is
.Q..
..Q.
q...
...q
Trying again...
Solved puzzle.
```

```
.q..
...q
q...
..q.
Solved 100.0%
```

Listing 2: Output for n = 4 using Simulated Annealing

```
Number of queens (1<=nqueens<100): 4
Algorithm: (1) Random search (2) Hill climbing (3) Simulated Annealing
    (4) Genetic Algorithm: 3

Initial state:
..q.
Q...
.Q..
.Q..
Solved puzzle.
..q.
q...
...q
.q..
Solved 100.0%
```

Listing 3: Output for n = 4 using Genetic Algorithm

```
Number of queens (1<=nqueens<100): 4
Algorithm: (1) Random search (2) Hill climbing (3) Simulated Annealing
    (4) Genetic Algorithm: 4

Initial state:
..Q.
...Q
...Q
q...
0 Solved puzzle.
.q..
...q
q...
..q.
Solved 100.0%
```

Listing 4: Output for n = 16 using Simulated Annealing: the algorithm fails to find a solution this time.

```
Number of queens (1<=nqueens<100): 16
Algorithm: (1) Random search (2) Hill climbing (3) Simulated Annealing
    (4) Genetic Algorithm: 3

```

```
Initial state:
......q.........
...q............
Q...............
....q...........
...........Q....
............Q...
..q.............
..........Q.....
.............Q..
.........Q......
.Q..............
..........Q.....
.Q..............
.............Q..
Q...............
...............Q
Puzzle not solved. Final state is
........q.......
...........q....
.q..............
.......q........
Q...............
..........q.....
.............q..
..q.............
..............q.
.........q......
....q...........
Q...............
.....q..........
...............q
............q...
......q.........
Solved 0.0%
```

Program files

Listing 5: nqueens.c

```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

```

```c
#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 */
int solved = 0;

typedef struct board {
  int queens[MAXQ];
  int fitness;
} Board;


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));
  srandom((unsigned int)time(NULL));
  srand((unsigned int)time(NULL));
}

/* 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, */
  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.
 */
```

```c
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;
}

int countConflictsOnBoard(int *x) {
  int cnt = 0;
  int queen, other;
  for (queen=0; queen < nqueens; queen++) {
    for (other=queen+1; other < nqueens; other++) {
      if (inConflict(queen, x[queen], other, x[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 evaluateBoard(int *x) {
  return (nqueens-1)*nqueens/2 - countConflictsOnBoard(x);
}

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

/* 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. ");
    solved++;
  }
  printf ("Final state is");
  printState();
}

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

void calculateNeighborsHC(int *neighbors, int queen) {
  for (int pos = 0; pos < nqueens; pos++) {
    moveQueen(queen, pos);
    neighbors[pos] = evaluateState();
  }
```

```
}

int bestNeighbor(int queen) {
  int neighbors[MAXQ];
  int best = -1;
  int sameValue[MAXQ], sizeSameValue = 0;
  calculateNeighborsHC(neighbors, queen);
  for (int i = 0; i < nqueens; i++) {
    if (neighbors[i] > best) {
      best = neighbors[i];
      sizeSameValue = 0;
    }
    if (neighbors[i] == best) {
      sameValue[sizeSameValue] = i;
      sizeSameValue++;
    }
  }

  return sameValue[random() % sizeSameValue];
}

void hillClimbing(int limit) {
  int queen, iter = 0;
  int optimum = (nqueens-1)*nqueens/2;
  int eval;

    /* generate new, better state: for each queen do ...*/
  for (queen=0; queen < nqueens; queen++) {
    int newpos;
    /* change to one of the best neighbors */
    newpos = bestNeighbor(queen);
    moveQueen(queen, newpos);
  }

  eval = evaluateState();
  if (eval == optimum) {
    printf ("Solved puzzle. ");
    solved++;
  }
  else
    printf ("Puzzle not solved. Final state is");

  printState();

  // Try again using the final state as initial state?
  if (eval != optimum && limit > 0) {
    initiateQueens(1);
    hillClimbing(limit - 1);
  }

```

```
}

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

float T(int t) {
  return 10000000 * pow(0.95, t);
}

double P(int deltaE, float temperature) {
  return exp(deltaE / temperature) * ((double) RAND_MAX + 1.0);
}

void simulatedAnnealing() {
  unsigned int t;
  float temperature;
  int nextQueen, nextPos, oldPos;
  int current, deltaE, eval;
  int optimum = (nqueens-1)*nqueens/2;

  for (t = 0; TRUE; t++) {
    int new;
    temperature = T(t);

    if (temperature <= 0)
      break;

    // Picking a random neighbor
    nextQueen = random() % nqueens;
    nextPos = random() % nqueens;
    oldPos = queens[nextQueen];
    while (oldPos == nextPos)
      nextPos = random() % nqueens;

    current = evaluateState();
    moveQueen(nextQueen, nextPos);
    new = evaluateState();
    deltaE = new - current;

    // Stop if a solution was reached
    if (new == optimum){
      eval = new;
      break;
    }

    // Undo the move if it's not a good move, but only with probability
         1 - e^(deltaE / temperature)
    if (deltaE <= 0 && rand() > P(deltaE, temperature))
      moveQueen(nextQueen, oldPos);
  }

```

```c
  if (eval == optimum) {
    printf ("Solved puzzle. ");
    solved++;
  }
  else
    printf ("Puzzle not solved. Final state is");

  printState();

}

int timeToTemperature(float temp){
  int cont = 0;

  for (int cont; TRUE; cont++)
    if (T(cont) <= temp)
      return cont;
}

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

void printBoard(Board *b) {
  printf("Address: %p\nBoard: ", b);
  for (int i = 0; i < nqueens; i++)
    printf("%d ", b->queens[i]);
  printf("\nFitness: %d\n\n", b->fitness);
}

void reproduce(int *x, int *y, Board* child) {
  int c = 1 + rand() % (nqueens - 1);
  for (int i = 0; i < c; i++)
    child->queens[i] = x[i];

  for (int i = c; i < nqueens; i++)
    child->queens[i] = y[i];

  child->fitness = evaluateBoard(child->queens);
}

void swapPermutation(int *x, int left, int right) {
  int aux;
  aux = x[left];
  x[left] = x[right];
  x[right] = aux;
}

void shiftPermutation(int *x, int left, int right) {
  for (int i = right; i > left + 1; i--)
    swapPermutation(x, i - 1, i);
}
```

```
void reversalMutation(int *x, int left, int right) {
  while (left < right)
    swapPermutation(x, left++, right--);
}

void scrambleMutation(int *x, int left, int right) {
  if (right != left)
    for (int i = left; i < right + 1; i++) {
      int r = left + rand() % (right - left);
      swapPermutation(x, i, r);
    }
}

void mutate(int *x) {
  int left = rand() % nqueens;
  int right = rand() % nqueens;

  if (left > right) {
    int aux = right;
    right = left;
    left = aux;
  }

  switch(rand() % 4) {
    case 0: shiftPermutation(x, left, right); break;
    case 1: swapPermutation(x, left, right); break;
    case 2: reversalMutation(x, left, right); break;
    case 3: scrambleMutation(x, left, right); break;
  }
}

void populate(Board *pop) {
  int q;
  for (q = 0; q < nqueens; q++)
    pop->queens[q] = rand()%nqueens;

  pop->fitness = evaluateBoard(pop->queens);
}

int binarySearch(int *a, int size, int value) {
  int low = 0;
  int high = size - 1;

  while (low <= high) {
    int mid = (low + high) / 2;

    if (a[mid] >= value)
      high = mid - 1;

```

```
    else
      low = mid + 1;
  }
  return low;
}

int binarySearchBoard(Board *b, int size, int value) {
  int low = 0;
  int high = size - 1;

  while (low <= high) {
    int mid = (low + high) / 2;

    if (b[mid].fitness <= value)
      high = mid - 1;

    else
      low = mid + 1;
  }
  return low;
}

void insertSorted(Board *population, int size, Board new) {
  // Position for board to be inserted
  int pos = binarySearchBoard(population, size, new.fitness);

  population[size - 1] = new;

  // Shifting to the right position
  for (int i = size - 1; i > pos; i--) {
    Board aux = population[i - 1];
    population[i - 1] = population[i];
    population[i] = aux;
  }

}

// Returns the position of a certain (random) board. Boards with higher
     values
// have better odds to be selected.
int select(Board *population, int size, int ignore) {

  int accumulatedFitness[size];
  int r;

  accumulatedFitness[0] = population[0].fitness;
  for (int i = 1; i < size; i++)
    if (i != ignore)
      accumulatedFitness[i] = population[i].fitness +
          accumulatedFitness[i - 1];
```

```c
    else if (ignore != -1)
      accumulatedFitness[i] = accumulatedFitness[i - 1];

  r = rand() % (accumulatedFitness[size - 1] + 1);

  return binarySearch(accumulatedFitness, size, r);
}

int cmpfunc(const void * a, const void * b) {
   const void *left = (const void **) a;
   const void *right = (const void **) b;
   const Board *board1 = left;
   const Board *board2 = right;
   return ( (*board2).fitness - (*board1).fitness);
}

void isSorted(Board *b, int size) {
  for(int i = 0; i < size - 1; i++)
    assert(b[i].fitness >= b[i + 1].fitness);
}

void printPopulation(Board *population, int size) {
  printf("--------------------------------\nPOPULATION:\n");
  for(int i = 0; i < size; i++)
    printBoard(&population[i]);
}

#define MAXITER_GA 100
void geneticAlgorithm() {

  int size = 1000;
  Board population[size];
  int optimum = (nqueens-1)*nqueens/2;
  int iter = 0;
  int chance = 2;

  for (int i = 0; i < size; i++)
    populate(&population[i]);

  qsort(population, size, sizeof(Board), cmpfunc);

  while (evaluateState() != optimum && iter < MAXITER_GA) {
    for (int i = 0; i < size / 2; i++) {
      Board child;
      int index = select(population, size, -1);
      Board x = population[index];
      Board y = population[select(population, size, index)];

      reproduce(x.queens, y.queens, &child);

```

```c
      if (rand() % 1000 <= chance)
        mutate(child.queens);

      insertSorted(population, size, child);
      // isSorted(population, size);
    }
    for (int j = 0; j < nqueens; j++) {
        queens[j] = population[0].queens[j];
    }
    printf("%d ", iter);
    iter++;
  }

  if (iter < MAXITER_GA) {
    printf ("Solved puzzle. ");
    printState();
    solved++;
  }
  else printf("Solution not found\n");
}

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

int main(int argc, char *argv[]) {
  int algorithm;
  int repeat = 1;
  int v[11] = {5, 10, 20, 20, 46, 48, 67, 79, 93, 96, 100};
  int r;
  initializeRandomGenerator();

  // Board x, y, z;
  // Board boards[10];
  // populate(&x);
  // populate(&y);
  // nqueens = 11;
  // qsort(boards, 10, sizeof(Board), cmpfunc);
  // isSorted(boards, 10);

  // insertSorted(boards, 10, x);
  // isSorted(boards, 10);

  // insertSorted(boards, 10, y);
  // isSorted(boards, 10);

  // printf("Sorting working\n");

  if (argc == 2)
    repeat = atoi(argv[1]);

  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();

  for (int i = 0; i < repeat; i++) {
    initiateQueens(1);

    // printf("\nInitial state:");
    // printState();

    switch (algorithm) {
    case 1: randomSearch();    break;
    case 2: hillClimbing(7);   break;
    case 3: simulatedAnnealing(); break;
    case 4: geneticAlgorithm(); break;
    }
  }
  printf("Solved %.1f%%\n", ((float) solved / repeat) * 100);

  return 0;
}
```

Game of Nim

Program description

The aim of this program is to simulate the optimal plays of a game of Nim. Nim is a two-player game where the players take turns removing 1, 2 or 3 matches from a pile of n matches, with $n >= 3$. Whoever takes the last match loses the game. The program is designed to work with starting values of up to 100 matches, though it is easy to configure it for more matches.

Problem analysis

In order to implement the decision making involved in this game, the Negamax algorithm was implemented. The program uses the algorithm to remove the best amount of matches it can so as to win.

Program design

The program was based on a Minimax implementation. The Minimax version presented a few flaws which are fixed in the Negamax: repeating code and recursive methods returning different types of data. To fix the first of these issues the Negamax algorithm does a pretty good job because of the essence of the algorithm (the -1 multiplication). The second issue, however, was fixed by creating a new type, Move, to keep the information together. This allowed the recursive method to return the needed information, the move and the evaluation of such move, in a consistent way. The resulting code is much cleaner and easier to understand than the Minimax version.
Because the Negamax algorithm on its own resulted in repeating the same calculations several times, a transposition table was also implemented, greatly improving the program's performance. Moreover, note that the new type created, Move, also helps to store the information in the table in a clean, consistent way.

Program evaluation

The program was verified by comparing the output of the Minimax version with the program's output. To do that, a shell script was created to run both programs with values ranging from 3 to 30 and compare the outputs. The script prints a message each time the outputs are different.
By running the script in Listing 6 no output was given, so the program works as desired, at least for the values tested. In terms of efficiency, the program also gives great results: even starting with 100 matches the output appears instantaneously on the screen.

Listing 6: checkNim.sh

```
for i in {3..30}; do
	./nim $i > nim.out
	./nimNegamax $i > nimNegamax.out
	cmp -s nim.out nimNegamax.out
	if [ $? != 0 ]; then
		echo "FAIL: n = $i"
	fi
done
rm nim.out
rm nimNegamax.out
```

Program output

The program outputs the simulation of a game using optimal plays. The following outputs are simulations of two games: one where Max wins (Listing 7) and one where it loses (Listing 8).

Listing 7: Output for n = 10

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

Listing 8: Output for n = 13

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

Program files

nimNegamax.c

Listing 9: nimNegamax.c

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

#define MAX 0
#define MIN 1
#define MAXMATCHES 100

#define INFINITY 9999999

typedef struct move {
  int move;
  int valuation;
} Move;

Move transpTable[2][MAXMATCHES + 1];

Move negaMax(int state, int turn) {
  Move best;
  best.valuation = INFINITY;
  best.move = -10;

  if (state == 1) {
    best.valuation = 1;
    return best;
```

```
  }

  if (transpTable[turn][state].move == -1){
    for (int move = 1; move <= 3; move++) {
      if (state - move > 0) {
        Move m = negaMax(state - move, 1 - turn);
        if (m.valuation != INFINITY)
          m.valuation *= -1;

        if (m.valuation < best.valuation) {
          best.valuation = m.valuation;
          best.move = move;
        }
      }
    }
    transpTable[turn][state] = best;
  }
  else {
    // printf("HELLO\n");
    best = transpTable[turn][state];
  }
  return best;
}

void initialize(int state){
  Move def;
  def.move = -1;
  def.valuation = -1;
  for (int i = 0; i <= state; i++){
    transpTable[MAX][i] = def;
    transpTable[MIN][i] = def;
  }
}

void playNim(int state) {
  int turn = 0;

  initialize(state);

  while (state != 1) {
    Move action = negaMax(state, turn);
    printf("%d: %s takes %d\n", state,
           (turn==MAX ? "Max" : "Min"), action.move);
    state = state - action.move;
    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;
}
```