Artificial Intelligence 1
Lab 2: Local and Adversarial Search

Albert Dijkstra (s3390071)
Niels Bugel (s3405583)
Edward Boere (s3005690)

LC[7] Team Supreme

May 18, 2018

Programming assignments

Exercise 1: N-queens problem

Hill climbing

Problem analysis

For hill climbing we need to make a board so we can see the amount of attacking pairs for each position we can move a queen to. Then we need to determine the lowest value of this board and move our queen to this value. We keep doing this until we cannot find a value lower than the current value.

Program design

We make a matrix with at each index the amount of conflicts if we move the queen in that column to that index. We then determine the minimum value of this matrix. If there are multiple indexes with this value, we choose a random one. Once the index has been chosen, we move the queen in the corresponding column to this index. Then we update the matrix with the new configuration and keep repeating this process.

Program extension

The program can get stuck at plateaus. To fix this, we allow a certain amount of sideways steps. This way the program will not get stuck on most plateaus.

Program evaluation

For each of these we used 5000 iterations:

n = 12

output: correct solution
real 0m5.543s
user 0m0.007s
sys 0m0.000s

n= 25

output: correct solution
real 0m3.171s
user 0m0.129s
sys 0m0.000s

n = 99

output: correct solution
real 0m43.710s
user 0m41.502s
sys 0m0.000s

valgrind for n=25:

==2623==
==2623== HEAP SUMMARY:
==2623== in use at exit: 0 bytes in 0 blocks
==2623== total heap usage: 28 allocs, 28 frees, 4,748 bytes allocated
==2623==
==2623== All heap blocks were freed – no leaks are possible
==2623==
==2623== For counts of detected and suppressed errors, rerun with: -v
==2623== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

Questions

1. If we run the hill climbing program several times using different amounts of queens, most of the time it doesn't find a solution.

2. The hill climbing fails because it can get stuck at two different points. If the program finds a local minimum it does not look any further and if the program hits a plateau it will also stop.

3. We can fix our algorithm so that it will not get stuck on a plateau by adding the ability to do a certain amount of sidesteps.

4. The bold numbers are the amount of queens, the number behind it is the success rate

1	100	**21**	100	**41**	100	**61**	100	**81**	100
2	0	**22**	100	**42**	100	**62**	100	**82**	100
3	0	**23**	100	**43**	100	**63**	100	**83**	100
4	100	**24**	100	**44**	100	**64**	100	**84**	100
5	100	**25**	100	**45**	100	**65**	100	**85**	100
6	100	**26**	100	**46**	100	**66**	100	**86**	100
7	100	**27**	100	**47**	100	**67**	100	**87**	100
8	100	**28**	100	**48**	100	**68**	100	**88**	100
9	100	**29**	100	**49**	100	**69**	100	**89**	100
10	100	**30**	100	**50**	100	**70**	100	**90**	100
11	100	**31**	100	**51**	100	**71**	100	**91**	100
12	100	**32**	100	**52**	100	**72**	100	**92**	100
13	100	**33**	100	**53**	100	**73**	100	**93**	100
14	100	**34**	100	**54**	100	**74**	100	**94**	100
15	100	**35**	100	**55**	100	**75**	100	**95**	100
16	100	**36**	100	**56**	100	**76**	100	**96**	100
17	100	**37**	100	**57**	100	**77**	100	**97**	100
18	100	**38**	100	**58**	100	**78**	100	**98**	100
19	100	**39**	100	**59**	100	**79**	100	**99**	100
20	100	**40**	100	**60**	100	**80**	100		

As we can see, our program always finds a solution. Except for two cases: 2 queens and 3 queens. The reason for this is that there are no configurations for those cases that satisfy the problem.
We ran each amount of queens 5 times by adding the following code:

```
        case 2:
            for (int j = 0; j < 100; j++)
            {
                nqueens = j;
                succes = 0;
                for (int i = 0; i < 5; i++)
                {
                    hillClimbing();
                    if(countConflicts() == 0)
                    {
                        succes++;
                    }
                }
                printf("with %d queens, succes: %d\n", nqueens,
                    succes);
            }
            break;
```

The result was a smoking computer and the number five being printed after every amount of queens.

Simulated annealing

Problem analysis

We used the pseudo code in the book. Based on this we wrote our own program.

Program design

We have a separate function that maps the time to temperature. In our main function we count the amount of conflicts of the current configuration and the amount of conflicts of the next configuration.
The difference here is that in the book, the pseudo code assumes that a higher value is better, however in our case, lower values are better. This is also the reason why our start temperatures are between 0 and 1.
Either that, or we would have to choose start temperatures such as 500 and instead of using $e^{\Delta E/temperature}$ we would have to use $e^{\Delta E*temperature}$.
Also, instead of using $\Delta E = next.VALUE - current.VALUE$ we use $\Delta E = currentConflicts - nextConflicts$

Program extension

Later on we added the function that half of the time it would pick the best option instead of a random one. More about this is in the answer of question 4. For this we used the function *climb* from hill climbing.

Program evaluation

For each of these we used 5000 iterations and start temperature 0.5:

n = 12

output: correct solution
real 0m3.290s
user 0m0.002s
sys 0m0.000s

n= 25

output: correct solution
real 0m3.044s
user 0m0.059s
sys 0m0.000s

n = 99

output: correct solution
real 0m18.305s
user 0m16.057s
sys 0m0.104s

valgrind for n = 25:

==2675==
==2675== HEAP SUMMARY:
==2675== in use at exit: 0 bytes in 0 blocks
==2675== total heap usage: 28 allocs, 28 frees, 4,748 bytes allocated
==2675== ==2675== All heap blocks were freed – no leaks are possible
==2675== ==2675== For counts of detected and suppressed errors, rerun with: -v
==2675== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

Questions

1. see report
2. With a linear function the program had trouble solving the problem. After some research we discovered that the following function was quite popular:

   $\frac{startTemp}{ln(time)}$

And indeed, our program found a solution more often with this function.

3. to test this we ran the same code snippet as we used for hill climbing except each amount of queens is tested 10 times instead of 5. Also we do not count the cases 2 and 3 as no solution found. With a start temperature of 0.01 it does not find a solution 100% after 20 queens The first time it found no solutions at all was after 51 queens. With a start temperature of 0.5 it does not find a solution 100% of the time after 13 queens. The first time it found no solutions at all was after 37 queens. With a start temperature of 0.99 it does not find a solution 100% of the time after 6 queens. The first time it found no solutions at all was after 37 queens.

   However for each of these temperatures, there are cases where it still finds all or most of the solutions. Based on this, we can conclude that temperatures closer to 0 are more likely to work for higher values of n, while start temperatures closer to 1 fail earlier. Note that this test used only 5000 iterations. If we use more iterations, the program is way more likely to find a solution.

4. The reason the algorithm doesn't work for n is higher than 10 is that at random the successor state is chosen (and then checked if it is better). The higher our n, the higher the chance that it will mostly choose successors that are (in the end) worse. We can make this better by instead of choosing its successor completely random, choosing 50% of the time the best case and otherwise a random one. This will always find the right solution. (Note!: if you make the amount of iterations very high (for example 50000) it will find for any n ¡ 99 a solution most of the time.

Genetic algorithm

Problem analysis

We can use divide and conquer to tackle this problem. There are several function that need to be implemented for a genetic algorithm. These functions are: a fitness function, a parent selection function, a reproduce function and a mutate function. We focused on these functions one by one.

Program design

The genetic algorithm needs a few functions. First it needs a fitness function that determines how good the current configuration is. We based this function on the amount of non attacking pairs. Then we need a random selection function. This function randomly selects two parents from the population and then selects the best of these two based on their fitness. Then we need a function that makes a child. This function has two parents and a random integer as input. Then the first part of parent 1 up until the random integer is copied to the child, the rest is from parent 2.

Program evaluation

For each of these outputs we used the following:

population size: $10 * amount\ of\ queens$
max iterations: 5000
mutate chance: 35%

8 queens

real 0m4.027s
user 0m0.035s
sys 0m0.000s

25 queens

real 0m6.459s
user 0m1.340s
sys 0m0.000s

99 queens

real 0m6.459s
user 0m1.340s
sys 0m0.000s

valgrind for n = 12

==2764==
==2764== HEAP SUMMARY:
==2764== in use at exit: 0 bytes in 0 blocks
==2764== total heap usage: 34,678 allocs, 34,678 frees, 1,666,496 bytes allocated
==2764==
==2764== All heap blocks were freed – no leaks are possible
==2764==
==2764== For counts of detected and suppressed errors, rerun with: -v
==2764== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

Even though this algorithm basically always finds a solution, it takes a very very long time when the number of queens is very high.

We see that the genetic algorithm is the slowest. Hill climbing and simulated annealing are quite similar, but simulated annealing is generally a bit faster. Simulated annealing also finds a solution more often.

Exercise 2: Game of Nim

Problem description

Nim is a simple two-player game. There exist many variations of the game. In this lab, we consider the following variation. We start with a pile of n matches, where $n \geq 3$. Two players, Max and Min, take turns to remove k matches from the pile, where $k = 1$, $k = 2$, or $k = 3$. The player who takes the last match loses. Max makes the first move.

Problem analysis

To decide on the optimal move, we look for which move maximizes the worst-case scenario. This way, if the other agent plays suboptimal, our strategy will always result in our expected outcome, or a better one. We can use the MiniMax algorithm, as described in the book on page 116, to calculate the outcome of the game if both agents play optimally.

Program design

We only need one function negaMax which first checks whether it is a terminal. If this is the case it will return the value that is on the leaf. After that, it will go trough all possible options of the amount of matches it can take of the table. And recursively calls the function again with the new values. After that it knows the best move and returns that.

Program extension

We add a transposition table to our program. For this we create an array of length 200. We initialize this array with 0s. This way we can easily check if the state has already been visited. If it has not been visited we put the move in this array. By doing this we prevent our program from checking states that have already been visited.

Program evaluation

We ran the program for the following inputs:

matches = 5:

5: Max takes 1
4: Min takes 3

1: Max looses

matches = 22:

22: Max takes 1
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

Valgrind for n = 74:

==2050==
==2050== HEAP SUMMARY:
==2050== in use at exit: 0 bytes in 0 blocks
==2050== total heap usage: 1 allocs, 1 frees, 1,024 bytes allocated
==2050==
==2050== All heap blocks were freed – no leaks are possible
==2050==
==2050== For counts of detected and suppressed errors, rerun with: -v
==2050== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

Programming questions

1. Use the utility $+1$ for a win by Max, and -1 for a win by Min (there cannot be a draw in Nim!). Consider the games with $n = 3$, $n = 4$, $n = 5$, and $n = 6$ matches. Who will (assuming optimal play) win which game? Explain why.

   $n = 3$, Max begins; he removes two matches from the pile ($k = 2$), now $n = 1$. The only legal move for Min is to remove one match from the pile ($k = 1$). This results in a win for Max.

   $n = 4$, Max begins; he removes three matches from the pile ($k = 3$), now $n = 1$. The only legal move for Min is to remove one match from the pile ($k = 1$). This results in a win for Max.

$n = 5$, Max begins; he cannot remove four matches from the pile, this is not a legal move. So there is no way for him to win the game next turn. If he removes one, two or three matches from the pile, the game will continue at $n = 4$, $n = 3$ or $n = 2$. The first two situations result in a win for the player next to make a move (as demonstrated above). For $n = 2$, Min is next. Min removes one match from the pile, and gives the turn to Max at $n = 1$. The only legal move is to remove one match from the pile, resulting in a win for Min.

$n = 6$, Max begins; he removes one match from the pile and gives the turn to Min with $n = 5$. As seen above, for $n = 5$, every move results in a loss. Max wins.

2. see code/programming report

3. With the current program, the higher the amount of matches, the longer we have to wait, because we need to check a lot of states.
   The transposition table helped a lot as it is now able to solve all games (up until 100), because we do not check the duplicate states.

Program files for exercise 1

nqueens.c

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

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

#define MAXQ 100
#define MAXITERATIONS 5000
#define POPULATIONSIZE 1000
#define MUTATECHANCE 35

#define FALSE 0
#define TRUE 1
#define MAXSW 100

#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, */
  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();
```

```
}

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

/* 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 climb(int * minConflicts, int * startConflicts, int ** costMatrix)
{
        *minConflicts = countConflicts();
        *startConflicts = *minConflicts;
        int counter = 0;

        //for loop that puts the conflict values in the matrix
        for (int i = 0; i < nqueens; i++)
        {
                int currentPos = columnOfQueen(i);
                for (int j = 0; j < nqueens; j++)
                {
                        //put a -1 at the position where we find a queen
                        if(currentPos == j)
                        {
```

```
                                costMatrix[i][j] = -1;
                        }
                        else
                        {
                                queens[i] = j;
                                costMatrix[i][j] = countConflicts();

                                //we found a new lowest value
                                if(costMatrix[i][j] < *minConflicts)
                                {
                                        *minConflicts = costMatrix[i][j];
                                        counter = 1;
                                }
                                else if(costMatrix[i][j] == *minConflicts)
                                {
                                        counter++;
                                }
                        }
                }
                //reset queens
                queens[i] = currentPos;

        }

        //select a random queen from the minimum values and move that
            queen
        counter = random() % (counter + 1);

        for (int i = 0; i < nqueens; i++)
        {
                for (int j = 0; j < nqueens; j++)
                {
                        if(costMatrix[i][j] == *minConflicts)
                        {
                                if(counter == 0)
                                {
                                        queens[i] = j;
                                }
                                counter--;
                        }

                        if(counter == -1)
                        {
                                break;
                        }
                }
        }
}

//function that implements hill climbing
```

```
void hillClimbing() {
  int** costMatrix = malloc(nqueens*sizeof(int*));
  for (int i = 0; i < nqueens; i++)
  {
          costMatrix[i] = malloc(nqueens*sizeof(int));
  }

  int minConflicts;
  int startConflicts;
  int sideways = MAXSW;

  do
  {
          climb(&minConflicts, &startConflicts, costMatrix);

          sideways = (minConflicts == startConflicts)? sideways - 1:
              MAXSW;
  } while(minConflicts < startConflicts || sideways > 0);

  //print solution and free everything
  printState();
  for (int i = 0; i < nqueens; i++)
  {
          free(costMatrix[i]);
  }
  free(costMatrix);
}

/**********************************************************/
//function that maps the time to temperature
float timeToTemperature(int t) {
        float startTemp = 0.5;
        return startTemp / (float)log(t+1);
}

void simulatedAnnealing() {
        float temperature;
        int next;
        int deltaE;
        int newConflicts;
        int startConflicts;
        int currentPos;

        //initiliaze cost matrix
        int** costMatrix = malloc(nqueens*sizeof(int*));
        for (int i = 0; i < nqueens; i++)
        {
                costMatrix[i] = malloc(nqueens*sizeof(int));
        }

```

```
	int minConflicts;

	for (int i = 0; i <= MAXITERATIONS; i++)
	{
		//have a 50% chance it will actually choose the best option
		if(0.5 < (double)rand() / (double)RAND_MAX)
		{
			climb(&minConflicts, &startConflicts, costMatrix);
		}
		else
		{
			//current conflicts
			startConflicts = countConflicts();


			temperature = timeToTemperature(i);

			//solution found
			if(temperature == 0 || startConflicts == 0)
			{
				printState();
				for (int i = 0; i < nqueens; i++)
				{
					free(costMatrix[i]);
				}
				free(costMatrix);
				return;
			}

			do
			{
				next = random() % (nqueens * nqueens);
			}while(!canMoveTo(next / nqueens, next % nqueens));

			//next state conflicts
			currentPos = columnOfQueen(next / nqueens);
			queens[next/nqueens] = next % nqueens;
			newConflicts = countConflicts();

			deltaE = startConflicts - newConflicts;
			//reset queen
			queens[next/nqueens] = currentPos;


			//determine if we go to the next state
			if(newConflicts <= startConflicts)
			{
				queens[next/nqueens] = next % nqueens;
			}
```

```c
					else if (exp(deltaE/temperature) > (double)rand()
					    / (double)RAND_MAX)
					{
						queens[next/nqueens] = next % nqueens;
					}
				}

		}
		printState();
		for (int i = 0; i < nqueens; i++)
		{
			free(costMatrix[i]);
		}
		free(costMatrix);
}

/**********************************************************/
//fitness function based on the amount of non attacking pairs
int fitness(int * currentQueens)
{
		for (int i = 0; i < nqueens; i++)
		{
			queens[i] = currentQueens[i];
		}
		return nqueens * (nqueens - 1)/ 2 - countConflicts();
}

//create a child based of two parents
int * reproduce(int * x, int * y, int n)
{
		int *child = malloc(nqueens * sizeof(int));

		for (int i = 0; i < n; i++)
		{
			child[i] = x[i];
		}

		for (int i = n; i < nqueens; i++)
		{
			child[i] = y[i];
		}

		return child;
}

//select two parents and select the best out of the two based on fitness
int * randomSelection(int * currentPopulation[POPULATIONSIZE], int
    totalCurrentPQ)
{
		int rand1 = random() % (nqueens*10);
```

```c
        int rand2;
        do
        {
                rand2 = random() % (nqueens*10);
        }while(rand1 == rand2);

        if(fitness(currentPopulation[rand1]) >
            fitness(currentPopulation[rand2]))
        {
                return currentPopulation[rand1];
        }
        return currentPopulation[rand2];
}

//do a random mutation to the child
int * mutate(int * child)
{
        if(random() % 100 <= MUTATECHANCE)
        {
                int rand = random() % nqueens;
                child[rand] = random() % nqueens;
        }
        return child;
}

void genetic()
{
        int * newPopulation[POPULATIONSIZE];
        int * currentPopulation[POPULATIONSIZE];
        int totalNewPQ = 0;
        int totalCurrentPQ = 0;

        //initialize some random population
        for (int i = 0; i < nqueens*10; i++)
        {
                currentPopulation[i] = malloc(nqueens * sizeof(int));
                for (int j = 0; j < nqueens; j++)
                {
                        currentPopulation[i][j] = random()%nqueens;
                }
                totalNewPQ += fitness(currentPopulation[i]);
        }


        for (int i = 0; i < MAXITERATIONS; i++)
        {
                totalCurrentPQ = totalNewPQ;
                totalNewPQ = 0;

                //go through the population
```

```
for (int j = 0; j < nqueens*10/2; j++)
{
	//set from two parents two new childs
	int * x;
	int * y;
	int * child1;
	int * child2;
	int n = random()%(nqueens - 2) + 1;
	x = randomSelection(currentPopulation, totalCurrentPQ);
	do
	{
		y = randomSelection(currentPopulation, totalCurrentPQ);
	}while(x == y);

	//reproduce
	child1 = reproduce(x, y, n);
	child2 = reproduce(y, x, n);

	//mutate
	mutate(child1);
	mutate(child2);

	if(i != 0)
	{
		free(newPopulation[2*j]);
		free(newPopulation[2*j+1]);

	}

	//set it in the new population and return it if they are the final solution
	newPopulation[2*j] = child1;
	newPopulation[2*j+1] = child2;

	if(fitness(child1) == nqueens * (nqueens - 1)/ 2)
	{
		for (int k = 0; k < nqueens; k++)
		{
			queens[k] = child1[k];
		}
		printState();
		for (int i = 0; i < nqueens*10; i++)
		{
			free(currentPopulation[i]);
			free(newPopulation[i]);
		}
		return;
	}
```

```
                if(fitness(child2) == nqueens * (nqueens - 1)/ 2)
                {
                    for (int k = 0; k < nqueens; k++)
                    {
                        queens[k] = child2[k];
                    }
                    printState();
                    for (int i = 0; i < nqueens*10; i++)
                    {
                        free(currentPopulation[i]);
                        free(newPopulation[i]);
                    }
                    return;
                }
                totalNewPQ += fitness(child1) + fitness(child2);



            }
            //set currentPopulation to the new population
            for (int j = 0; j < nqueens*10; j++)
            {
                for (int k = 0; k < nqueens; k++)
                {
                    currentPopulation[j][k] =
                        newPopulation[j][k];
                }

            }
        }
        //find the best one of the population
        int maxPQ = fitness(currentPopulation[0]);
        int index = 0;
        for (int i = 1; i < nqueens*10; i++)
        {
            int pq = fitness(currentPopulation[i]);

            if(pq > maxPQ)
            {
                maxPQ = pq;
                index = i;
            }
        }

        //copy the best solution
        for (int i = 0; i < nqueens; i++)
        {
            queens[i] = currentPopulation[index][i];
        }

```

```
        //print solution and free everything
        printState();
        for (int i = 0; i < nqueens*10; i++)
        {
                free(currentPopulation[i]);
                free(newPopulation[i]);
        }
}

int main(int argc, char *argv[])
{
  srand ((unsigned int)time(NULL));
  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: ");
    scanf ("%d", &algorithm);
  } while ((algorithm < 1) || (algorithm > 4));

  initializeRandomGenerator();

  initiateQueens(1);

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

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

  return 0;
}
```

Program files for exercise 2

nim.c

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

#define MAX 0
#define MIN 1

#define INFINITY 9999999

int negaMax(int state, int color, int * bestMove, int transpositionTable[200])
{
	//is terminal
	if (state == 1)
	{
		//psuedocode states return -color, but that doesn't work,
		//if we return -1 it works exactly the same (up untill 20) as the algorithm in the given code
		return -1;
	}

	//return from the transpositionTable if it excist
	if(transpositionTable[100 + color*state] != 0)
	{
		int winOrLoose = - 1 * negaMax(state-transpositionTable[100 + color*state], -1 * color, bestMove, transpositionTable);
		*bestMove = transpositionTable[100 + color*state];
		return winOrLoose;
	}

	int v = -INFINITY;
	int currentBestMove = 1;
	for (int move = 1; move <= 3; move++)
	{

		if (state - move > 0)
		{
			int m = - 1 * negaMax(state-move, -1 * color, bestMove, transpositionTable);
			if(m > v)
			{
				currentBestMove = move;
				v = m;
			}
		}
	}
```

```c
        transpositionTable[100 + color*state] = currentBestMove;
        *bestMove = currentBestMove;
        return v;
}

void playNim(int state)
{
        int turn = 0;
        int transpositionTable[200] = { 0 };

        while (state != 1)
        {
                int action = 1;
                negaMax(state,(turn==MAX) ? 1 : -1, &action,
                    transpositionTable);

                //reset transposition table
                for (int i = 0; i < 200; i++)
                {
                        transpositionTable[i] = 0;
                }

                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;
}
```