AI 1: Lab session 2

Local and Adversarial Search

Instructions

- Formulate your answers clearly and always provide a motivation for your answer.
- Read the book and use the information from the lectures. Read the questions carefully and feel free to ask for clarifications if something is not clear.
- You can use the helpdesk email `airug1718@gmail.com` to ask questions. Make sure you read the instructions to contact the helpdesk provided on Nestor. Only emails sent according to the instructions will be replied to.
- For the programming assignments, always supply all relevant (self-written) source codes such that the teaching assistants can test them. It is not allowed to use code supplied by others. If we suspect plagiarism, the exam committee will immediately be notified!

Rules for submission

- Submit a digital version of your team's report through Nestor (via the Submissions section). Reports need to be written in LaTeX. A template is available on Nestor.
- Submit your report by the deadline provided on Nestor. Deadlines are strict.
- Supply the names and student numbers of all members of your team. Also clearly write down your Learning Community.

Grading Every session is graded by the teaching assistants. The grade of this lab assignment will count for 15% of your final grade. Note that the deadlines are strict: we subtract 2^{n-1} grading points for a report that is n days late.

Programming assignment 1: N-queens problem

A classic problem that lends itself for local search algorithms is the N-queens problem. In the first programming assignment, we try to tackle this problem using hill climbing, simulated annealing, and a genetic algorithm.

Download from Nestor the file `nqueens.c`. This file is the starting point for this exercise. The program uses a complete state algorithm, i.e. N queens are placed on an $N \times N$ chess board, one queen per row. We introduce the rule that a queen can only move horizontally within her row. The state is represented by the one-dimensional array queens. The number of queens is stored in the variable `nqueens`. We do not consider cases with $N > 100$.

```
#define MAXQ 100
int nqueens;      /* number of queens: global variable */
int queens[MAXQ]; /* queen at (r,c) is represented by queens[r] == c */
```

The state is encoded as follows. The queen that corresponds with row `r` is in column `queens[r]`. Note that the rows and columns are numbered `0..nqueens-1`.

Some functionality has already been implemented for you. Before you start, study the code in `nqueens.c`. Compile the program as follows:

```
gcc -Wall -o nqueens nqueens.c -lm
```

1. Hill climbing Once you succeeded to compile the program, you are ready to run it. A random search has already been implemented. Of course, that is a bad method, and will often fail. Still, it is a good idea to study the routine `randomSearch()` and use it as a skeleton for the implementation of better local search algorithms.

Implement yourself the hill climbing algorithm from the textbook. In the hill climbing process, you will often get in the situation that several neighboring states evaluate to the same best value. In that situation, you may choose randomly between the best choices. You can use the standard C-library function `random()` for this. This function returns an integer random number that is at least 0. So, if you want to choose a random value from the integer interval `[a,b)`, then you can do that by using `choice=a + random() % (b-a)`.

Note that the random generator `random()` is a so-called pseudo-random generator. For every invocation of the program, the sequence of 'random' numbers generated by `random()` is the same (but appears to be random). This is quite convenient, when you are debugging your program. However, in the 'production phase', you want more random behaviour. You can do this by 'seeding' the random generator with some initial value. A standard trick is to use the cpu-time (which will be different at each invocation of your program) as the seed for the random generator. You can do this with:

```
srand ((unsigned int)time(NULL));
```

Once you have implemented the algorithm, answer the following questions.

1. Run the algorithm several times, using different number of queens. Does the algorithm usually solve the problem?
2. In which situations does the algorithm fail to solve the problem?
3. What can you do to improve the algorithm? Implement your suggestions for improvement.
4. Make a table and/or plot showing the success rate versus number of queens of your modified code.

2. Simulated annealing We try to solve the N-queens problem using simulated annealing. You can find pseudo-code for this algorithm in the textbook.

1. Start by implementing a concrete C implementation of this pseudo-code. You probably need to modify the algorithm to make it suitable for the problem at hand. Make note of all your design decisions in your report.

2. Define a suitable formula for the temperature as a function of time:

   $$T(t) = ...$$

   Here, T denotes temperature and t denotes time. As a first try, you may choose a linear function. What is the effect of this function on the quality of the algorithm? Implement your function in a C function called `timeToTemperature()`. Motivate your choices in your report.

3. Run the algorithm using varying start temperatures and number of queens. Does the algorithm (often/always) return a solution? What settings should be chosen for which problem size?

4. Probably, your program does not work very well for problem sizes with more than 10 queens. Why is that? Try to modify your code, such that it also works for larger problem sizes. You may use any trick/heuristic that you can come up with, as long as the search remains a local search.

3. Genetic algorithm Again, we try to solve the N-queens problem. This time we use a genetic algorithm. You are completely free to choose yourself how to implement this (and you can of course implement the pseudo-code proposed in the textbook), what heuristic(s) to use, and which population size, mutations and cross-overs to use. Motivate and record your design decisions in the report.

Which of the three methods (Hill climbing, simulated annealing, and genetic algorithms) works best for the N-queens problem (for varying values of N)?

Programming assignment 2: Game of Nim

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.

For example, consider a game with initially $n = 7$ matches. Max starts and takes two matches, so there are 5 matches left. Next, Min takes 3 matches, leaving 2. Now, of course, Max takes 1 match, and Min loses.

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.

2. On Nestor you will find the source code of a simple program that simulates two optimally playing Nim players. The code is kept as simple as possible. However, this does have the disadvantage that there is some unnecessary code-duplication:

(a) the routines `minValue` and `maxValue` are very similar.

(b) the first choice in the game tree (the routine `minimaxDecision`) returns a move, while deeper recursive calls return a valuation.

Make a negamax-version of the minimax algorithm that returns pairs (move + valuation). Make sure that your program plays the same strategy as the original program. Include the code in your report.

3. Run your program for a game with $n = 10$, $n = 20$, $n = 30$, $n = 40$, and $n = 50$ matches. What do you observe?

   Extend your program with a transposition table (see textbook, Section 5.3). You may assume that the game is not played with initially more than 100 matches. Run the modified program again for $n = 50$. Did this help? Include the code in your report.