AI 1: Lab session 2

Local and Adversarial Search

Instructions

1. This is a ***team assignment*** to be completed in a team of two or three students and consists of a Python coding element and a report.
2. Answer all questions and ***formulate your answers concisely and clearly***.
3. Remember ***it is not allowed to use code not developed by your team***. Doing so constitutes plagiarism. If you use external sources (videos, websites, discussion fora, etc.) to develop your code make sure to ***clearly refer to them in your report and in your code***. Failing to disclose these sources constitutes again an instance of plagiarism.
4. ***Reports need to be written in LaTeX*** and submitted as a pdf ***separately from the code***. A template is available on Brightspace (under 'Course overview'). The text should be ***authored by your team and your team only***.
5. Comply with the deadline provided on Brightspace. ***Deadlines are strict*** (penalties apply for late submissions, see below).
6. The code should be submitted in a ***separate zip file***.

Assessment The grade of this lab assignment will counts for ***10% of your final grade***. We ***subtract*** 2^{n-1} grade points for a submission that is between $n-1$ and n days late ($n \geq 1$).

Questions? Get in touch with us! You can do so during labs, tutorials, using the discussion boards on Brightspace or the helpdesk email `ai12023@rug.nl`. Make sure you read the instructions to contact the helpdesk that are provided on Brightspace (under 'Course overview'). Only emails sent according to the instructions will be replied to.

1 Programming assignment 1: N-queens problem [60 pts]

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 Brightspace the file `nqueens.py`. 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 column. We introduce the rule that a queen can only move vertically within her column. The state is represented by the one-dimensional array `board`. The number of queens is found by getting the size of the array (`len(board)`). We do not consider cases with $N > 100$.

States are encoded as follows. The queen that corresponds with column `c` is in row `board[c]`. Note that the rows and columns are numbered `0..len(board)-1`.

Some functionality has already been implemented for you. Before you start, study the code in `nqueens.py`. Run the program as follows (with `0 < x <= 100`):

```
python3 nqueens.py x
```

1.1 Hill climbing [15 pts]

Once you've taken a quick look at the existing code you are ready to start. 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 `random_search()` 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 Python package `random` for this, in particular the function `random.randint(a, b)`. This function returns an integer random number that is in the range of `[a, b]`.

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

Questions:

a) Run the algorithm using $N = \{4, 8, 16, 32, 64\}$, averaging over ten runs per value of N . Does the algorithm usually solve the problem?

b) In which situations does the algorithm fail to solve the problem? Why?

c) What can you do to improve the algorithm? Implement one improvement.

d) Make a table or plot showing the success rate versus number of queens of your modified code. What do you observe?

1.2 Simulated annealing [20 pts]

We try now to solve the N-queens problem using simulated annealing. You can find pseudo-code for this algorithm in the textbook and on the slides.

Start by implementing this pseudo-code. You may need to slightly modify the algorithm to make it suitable for the problem at hand. Define a suitable formula for the temperature as a function of time:

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

Here, T denotes temperature and t denotes time. It may be beneficial for you to visualize your temperature function while tuning it. Implement your function in a function called `time_to_temperature()`.

Motivate your choices in your report and answer the following questions:

Questions:

a) How did you implement the algorithm?

b) What formula did you use for the `time_to_temperature()` function? Why?

c) Run the algorithm using $N = \{4, 8, 16, 32, 64\}$, averaging over ten runs. Does the algorithm always return a solution? Give the success rates of the different values for N. If you want you can increase the maximum number of iterations for this.

d) Change your temperature function such that the temperature drops too quickly. Then, change the temperature function such that is drops too slowly. Explain why the performance decreases for both cases. Support your explanations with experimental data.

1.3 Genetic algorithm [25 pts]

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 and discussed in Lecture 4), what heuristic(s) to use, and which population size, mutations and cross-overs to use.

Answer the following questions:

Questions:

a) Motivate and record your design decisions in the report. For example, discuss how parents are selected, children are generated and any mutations implemented.

b) Which of the three methods (Hill climbing, simulated annealing, and genetic algorithms) works best for the N-queens problem (for $N = \{4, 8, 16\}$, averaging over ten runs) and why does that algorithm outperform the rest?

2 Programming assignment 2: Game of Nim [30 pts]

Nim is a simple two-player game. There exist many variations of the game. In this lab, we consider the following variation, also discussed in Lecture 5. 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.

On Brightspace you will find the source code of a simple program that simulates two optimally playing Nim players. Run the program as follows (with `0 < x <= 100`):

```
python3 nim.py x
```

The code is kept as simple as possible. However, this does have the disadvantage that there is some unnecessary code-duplication:

- the routines `min_value` and `max_value` are very similar.
- the first choice in the game tree (the routine `minimax_decision`) returns a move, while deeper recursive calls return a valuation.

Make a negamax-version of the minimax algorithm that returns pairs, so a move and a valuation. In Python this is can be done by having the following return statement: `return value, best_move`.

Use the utility +1 for a win by Max, and -1 for a win by Min (there cannot be a draw in Nim!). Make sure that your program plays the same strategy as the original program.

Questions:

a) Consider the games with $n = 3$, $n = 4$, $n = 5$, and $n = 6$ matches. Who will (assuming optimal play) win which game? Explain why using the utility values.

b) Run your program for a game with $n = 10$, $n = 20$, $n = 30$ matches. What do you observe?

c) 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.

d) Run the modified program again for $n = 50$. Did the extension help? Why (not)?