Problem assignment 4
Due: Tuesday, October 3, 2017

Problem 1. Traveling salesman problem: Genetic algorithm

This problem investigates the same TSP problem we studied in Assignment 3.

The disadvantage of the simulated annealing algorithm is that at any point in time it keeps only one current configuration and that next step configurations are obtained using “local” changes. Thus, it may take a number of steps till one gets to explore good configurations. The genetic algorithm attempts to alleviate the problems by keeping a limited number of “current” configurations and by combining (more radically) two good quality solutions hoping that the combination will lead to a larger improvement in the quality.

Please familiarize yourself with the code in main4d.c and the methods next_gen() and pick_a_pair_random() in Population.cpp. You will find potentially useful comments there.

Collect results (fitness of the best individual in the last generation) for different settings of parameters of the genetic algorithm procedure. The procedure is included in main4d.cpp and uses the following five parameters:

1. Country – describes the problem (you do not have to change this one);
2. Number of generations to simulate;
3. Population size (max 500 or play with defs.h);
4. Mutation probability (a value between 0 and 1);
5. Survival rate (a value between 0 and 1). It determines how much of the old population is kept in the new population.

The default configuration is: Number of generations 300, Population size 300, Mutation probability 0.03, Survival rate 0.2.

Please report best configuration(s) found by varying:

- (a) Number of Generations in the range between 50 and 500 in increments of 50 while keeping all other parameters fixed at the default setting.
- **(b)** Population size in the range between 50 and 500 in increments of 50 with the rest of the parameters set to the defaults.

- **(c)** Mutation probability in the range between 0.00 and 0.25 in increments of 0.05 with the remaining parameters set to the default values.

- **(d)** Survival rate in between 0 and 1 in increments of 0.2 with the remaining parameters fixed at the defaults.

Please report your findings either in tables (one for each experiment) or graphs. Analyze the results and draw the conclusions. No submission of code is required for this part.

**Problem 2. Adversarial search.**

Consider the game search tree in the figure below

![Game Search Tree](image)

Assume the first player is the max player and the values at leaves of the tree reflect his/her utility. The opponent wants the same utility to be minimized.

**Part a.** Compute the minimax values for each node in the tree? What move should the first player choose? What is the solution path the rational players would play.

**Part b.** Assume we use alpha-beta algorithm to explore the game tree and we do this in the left-to-right order and determine the players strategies. List all nodes that are cut off from the tree and are never examined by the alpha beta procedure (see lecture notes for an example of how the alpha beta procedure works).

**Part c.** Assume we use alpha-beta algorithm but explore the tree in the right-to-left order. What nodes would not need to be examined by the alpha-beta algorithm and pruned away?
Problem 3. Tic-tac-toe-10 player

The goal of this assignment is to implement a program capable of playing Tic-tac-toe on the $10 \times 10$ board. The $3 \times 3$ version of the game is discussed in the textbook. Both $3 \times 3$ and $10 \times 10$ versions of the Tic-tac-toe are related to the old Japanese game of Go.

Rules. The game is played on a rectangular $10 \times 10$ board. There are two players, one plays crosses, the other plays zeroes. The players alternate their moves. The first one to achieve five of his marks in a row, a column or on a diagonal wins.

The player program you have to build will always play crosses. To test your program you may use board positions stored in the boards subdirectory. It is always X’s turn. We encourage you to create some interesting board positions yourself.

The program’s input is a current board position. The output is a number identifying the board position chosen by the program. A position on a $10 \times 10$ board, can be represented using two digits with values 0-9. The first digit indexes the row, the second digit the column. For example, the output '23' indicates the program chose to put its mark (an 'x') to the second row and the third column. Note that your program chooses only its next move.

To get you started we provide you with the most basic player in player.cpp. Please familiarize yourself with the code. There is one global variable used by the program: the variable board that holds a $10 \times 10$ array representing the board configuration. The array contains 'X', 'O' and '.' characters, with the obvious interpretation. The variable board is initialized by the board configuration on the standard input. The main function takes care of the initialization.

Tournament support. The support code written for this assignment lets you play two different programs against each other. The code takes care of switching between xs and os so both programs are assumed to be 'x' players. The support program is written in perl and the name of the script is driver.pl. By default the script iteratively calls playerA and playerB programs. To play two different programs you need to name one of them playerA and the other playerB. To play the same program against itself playerA and playerB programs are the same.

Part a. A tic-tac-toe player with a board evaluation heuristic.

Our objective is to create a decent player of the game. Ideally a player would search the complete game tree and compute the best move while considering the best (rational) responses of its opponent. However, in this and many other games we do not have the luxury of exploring the full game tree before making the move. To address this issue we consider a limited-depth exploration of the game tree, where nodes at the depth (or cutoff) limit are evaluated using a board evaluation heuristic. If the depth of the tree is k we refer to the procedure as to k-ply search.
The program `player.cpp` calls a minimax (alpha-beta) search procedure that takes a board position as its input, searches the game tree corresponding till level k and outputs the best next move according to the heuristic board evaluation function. The default level is k=2. The heuristic board evaluation function is given to you in file `heuristics.cpp`. Note that the heuristic function is used to evaluate the leafs of the tree only. Please compile the current player and run the two versions of the program against each other using the tournament `driver.pl`. Play at least 10 matches and report the table with results you have achieved in terms of wins, draws and losses of Player A (moves first) and Player B (moves second).

**Part b.** A tic-tac-toe player with improved heuristic evaluation.

One way we can improve a tic-tac-toe player is to improve its evaluation heuristic. The current heuristic function (file `heuristic.cpp`) analyzes the game configuration and identifies different line configurations of X and 0 symbols (in rows, columns and on diagonals). The line configurations found are assigned a qualitative label (win, good, threat, etc) which is then used to calculate the utility of the position. Propose modifications of the the file `heuristics.cpp` by trying: (1) different scoring of line configurations (2) addition of new line configurations. Experiment with different heuristics. Once you think your heuristic performs well copy it to file `heuristics2b.cpp`, compile it and play the new player against the player from part a. You should submit the new heuristic code in `heuristics2b.cpp`. Describe the changes you have made in the report including the intuition behind these changes.

Play the new player versus part-a-player 10 times starting the new player as Player A, and 10 times starting it as Player B. Report win, loss, draw statistics. If you think your new player is better than the part-a-player argue the case using statistics obtained in part a and part b.

**Play for credit.** The programs submitted by all students will be run against each other in a tournament. Each program will play each 10 times. Scores will be collected and winners announced. The winners will receive extra-credit for the assignment.