UMEÅ UNIVERSITY

Department of Computer Science
Assignment report
Generalised tic-tac-toe

Artificial intelligence
(5DV019), Fall 09
Assignment 1
first delivery

Artificial intelligence

Assignment 1 – Generalised tic-tac-toe

Tutor(s):
Johan Granberg

Petter Ericson, Eric Jönsson pettter, dv06ejn@cs.umu.se

~pettter, dv06ejn/edu/ai/lab1

September 25, 2009

Contents

1 Problem specification

The problem of search turns out to be a fundamental concept in artificial intelligence. Many problems can be formulated as a search problem, and a good part of the course is dedicated to search strategies and how to employ and improve upon them.

For this assignment, we were to create a game-playing (adversarial search) AI employing min/max search with alpha-beta-pruning and iterative deepening. We are pleased to report that we produced a fully-working implementation, see sections 3 and 4 of details of how it was done.

The complete problem specification can be downloaded from `http://www.cs.umu.se/kurser/5DV019/HT09/ttt.pdf`

2 Access and user manual

The program compiles and runs on (at least) the Linux- and Solaris machines available at the department of Computer Science at Umeå University. The source code can be fetched from ∼pettter, dv06ejn/edu/ai/lab1.

To compile, simply type `make`:

```
mega:~/edu/ai/lab1> make
mega:~/edu/ai/lab1>
```

This produces a game binary located in the `bin/` folder. To see how it works, type `bin/ttt`:

```
mega:~/edu/ai/lab1> bin/ttt
Usage: bin/ttt n m d [foo]
n: number of tic or tac in a row
m: size of game board (52 or less)
d: depth of AI search tree
if there is a fourth argument, the AI begins play
mega:~/edu/ai/lab1>
```

So for example, to play a typical four-in-a-row game on a seven by seven game board using an AI that thinks six moves ahead, type `bin/ttt 4 7 6`:

```
----------------
abcdefg
        a
        b
        c
        d
        e
        f
        g
----------------
X:
```

This presents an empty game board and asks you for the X coordinate to your preferred mark. After specifying the Y coordinate again, the AI "thinks" for a while and presents its move:

```
X: d
Y: d
----------------
abcdefg
        a
        b
        c
   X    d
    O   e
        f
        g
----------------
X:
```

If one should grow bored of waiting for the AI to react, you can send it the `SIGINT` signal, representing a software interrupt. The AI then places its mark at what it currently thinks is its best move and passes the turn.

The game continues until someone wins or the board becomes full at which point the program terminates:

```
abcdefg
        a
        b
    X O c
   XXO  d
    O   e
   O    f
  X     g
----------------
mega:~/edu/ai/lab1>
```

If one wishes to end the game prematurely, the user simply sends the `SIGINT` signal during his or her own turn.

3 System description

While the original plan for the system envisioned three modules; one being the user interface (the `src/ui` directory), the second the AI (`src/ai`) and the third being the game logic and main program (`main.c`), in the end the user interface was simplified and incorporated into the main program.

Further, the system was designed to have very loose coupling between the modules. In principle, only three callbacks are required for each player; an initiation, a method of returning the next move and a finish-up method.

4 Algorithm description

From an AI standpoint, there are only a few interesting topics worth mentioning among the techniques we used:

- Alpha-beta pruning

- Iterative deepening
- The heuristics function

For details on the heuristics function, (and indeed that is likely the most interesting read this paper has to offer) see section 5.

4.1 Alpha-beta pruning

Alpha-beta pruning can be considered an enhancement to the basic min/max search strategy, and so we shall refer to it directly when talking about game tree search. This implementation uses a similar search strategy as the one given in the course book, the main difference being that we pass a pointer to a "currently best move" in order to have the search function interruptable.

4.2 Iterative deepening

In order to make the AI interruptable and still produce a decent result,[1] we employed the commonly used technique of iterative deepening.

The search begins at one ply deep and a "best move" is recorded. It then continues to search deeper and deeper, overwriting its previous best move as it finds new ones. If the search reaches maximum depth or the search is interrupted, the last found "best move" is given.

5 The heuristics function

While the heuristics is not very complicated in concept, in practise it is rather cumbersome to implement. The basis of the algorithm is that every cell in the game grid has a value for each line it is part of, and the total heuristic is the sum of all of these values over the whole board.

For this application, a *line* is a number of adjacent cells, either horizontal, vertical, or on any diagonal with the same symbol. Also, the *degrees of freedom* of that line is defined as the number of free cells directly adjacent to the edge cells in the line's direction. I.e. if the line has two degrees of freedom, it can expand in either direction, if it has one, it can only expand in one direction, and if it has zero, it can not be expanded further.

Thus, the heuristic value of cell c for vertical lines is

$$\texttt{vert\_val}(c) = 1 + (l * (d) + d)$$

where l is the number of nodes in the vertical line c belongs to and d is the number of degrees of freedom of that line. Of course, the heuristic values are reversed when computing the opponents' cells. There are some obvious drawbacks to this heuristic. For example, no regard is given to whatever lies beyond the closest cell to the end of each line. However, it is fast to compute and empirical tests have determined it to work well enough to beat both authors.

[1] And in all fairness, to earn the extra 33 points (a whooping 3.3% of the total score) on the final exam.

6 Limitation of the solution

The program suffers from two big disadvantages – the lack of a proper user interface being one. A mouse-driven event UI has the obvious advantage of not having to specifying coordinates, (something that becomes particularly tiresome at large game areas) you can simply click where you want your mark to appear. For this assignment however, we deemed a proper user interface to be more trouble than it's worth, we much preferred to have a (fast!) working solution.

The second limitation is the lack of heuristics inheritance. Given a board of heuristic values and a next move, we'd like to be able to calculate the difference in heuristics value on the current board instead of generating it all again from scratch. We're not quite sure on how to actually do this, but we believe it's both possible and potentially very fast on larger boards.

7 Problems and reflections

Starting the assignment, we had a list of features we wanted to implement. These include:

- Multi-threading. We wanted to take advantage of the dual- and quad-core computers available today, but failed to find a good parallel algorithm for alpha-beta search or board evaluation. As such, the program remains single-threaded.
- Busy AI. The best chess players anticipate the opponents moves and doesn't sit idle while the opponent is thinking. We would have liked to have the AI consider possible moves even during the opponents turn in order to have it become even faster. Alas, it was not to be.

During the assignment, we also encountered the usual pitfalls commonly found when coding C: mysterious segmentation faults, jumps into invalid memory, intimate affairs with the GNU debugger during the long hard nights, and so on.

8 Test case

To demonstrate that the AI beats a moderate human player on four-in-a-row on a 7x7 board, the appendix includes a game log.