Project 1 - Adversarial Search – Artificial Intelligence

Teacher: Stefán Ólafsson

January 26, 2023

Use the Piazza page or Discord if you have any questions or problems with the assignment. **Start early**, so you still have time to ask in case of problems!

This assignment is supposed to be done in a group of 2-3 students; up to 4 is allowed. Note that everyone in the group needs to understand the whole solution even though you may distribute the implementation work.

Time Estimate

16 hours per student in addition to the time spend in the labs, assuming you work in groups of 2-3 students, did the previous lab assignments and and attended the lectures on search and adversarial search or worked through chapters 2, 3 and 6 in the book.

Problem Description

Implement an agent that is able to play the game of Knight-Through. This game is a somewhat simple variant of chess.

The game is played on an grid of width W ($3 \leq W \leq 10$) and height H ($4 \leq H \leq 10$). The initial state is setup such that both players have two rows of identical pieces of their respective color. White's pieces are on rows 1 and 2 while black's pieces are on rows $H-1$ and H. The two players "white" and "black" take turns in moving one of their pieces. White moves first.

Pieces can move onto an empty square like knights in chess, that is, they can move two squares forward and one to the side, or one square forward and two to the side. However, they can not capture opponents pieces like this. Pieces can capture an opponents piece by moving one square diagonally forward onto a square containing an opponents piece. Pieces can never move backwards. That is, white's pieces will move up the board (increase in the y coordinate) and black's pieces will only move down the board (decrease on the y coordinate).

The goal of the game is to advance any one piece to the opposite side of the board (as if to promote a pawn in chess). The game ends if one of the players has reached his goal or if the player whose turn it is does not have any legal moves. This can happen if the player does not have any pieces left, or if all their pieces are in positions where they cannot move. The game ends in a draw if none of the players wins.

For the purpose of communicating the moves to the game simulator (and the opponent), the legal moves are called `(move x1 y1 x2 y2)` meaning that the piece at x1,y1 is moved to x2,y2. x1,x2 are integers between 1 and W. y1,y2 are integers between 1 and H.

Tournament

Your agents will be pitted against each other in a tournament on different sizes of the game and with different time constraints, to see how will the agents can play the game. Bonus points will be awarded for the best agents.

Choosing the Right Programming Language for the Task

The implementation part of this project can be done in any language you like. We provide some starter code in Python and Java similar to the previous labs.

The code that does the search will run very often (up to millions of times for selecting a single move) and therefore needs to be fast. My own experiments with this assignment suggest that an implementation in Java is about 50 times faster on the larger environments than an implementation of the same state space model and search algorithm in Python. It is very hard to make up for this difference by improving the implementation.

Tasks

1. Develop a model of the environment. What constitutes a state of the environment? What is a successor state resulting of executing an action in a certain state? Which action is legal under which conditions? (Note that the board size is flexible and your agent should be able to handle games with different board sizes within the given restrictions for width and height).

2. Implement the model and connect it with the agent to keep track of the current state of the game. This should allow you to make an agent that plays random legal moves (without any search). **Test the state space model extensively! Bugs here will be very hard to find when you implement the search.** Make sure that all legal moves, but only legal moves are generated in various different situations. Make sure that the successor states are correct and that the original state stays unchanged when computing the successor state.

3. Implement a state evaluation function (heuristic) for the game. You can start with the following simple evaluation function for white (and use the negation for black):

   - 100, if white won the game
   - 0, for a draw
   - -100, if white lost the game
   - <*distance of most advanced black piece to row 1*> - <*distance of most advanced white piece to row H*>, for non-terminal states

4. Implement iterative deepening alpha-beta search and use this state evaluation function to evaluate the leaf nodes of the tree.

5. Keep track of and output the number of state expansions (total and per second), current depth limit of your iterative deepening loop and run-time of the search for each iteration of iterative deepening and in total. These numbers are very useful to compare with others or between different version of your code to see whether your search is fast (many nodes per second) and the pruning works well (high depth, but fewer expanded nodes).

6. Make sure you handle the time limits correctly. For that, your search must to stop when the time (play clock) is up. It should then return the best move found in the last search that finished (typically the previous iteration of iterative deepening), otherwise it runs the risk of returning a worse move simply because the search was interrupted before it looked at the best move. An easy way to stop the search is to

   (a) periodically check whether the time is up during the search and if it is, raise an error / throw an exception (e.g., `raise TimeoutError` in Python or `throw new TimeoutException()` in Java).

   (b) catch the exception/error outside of the iterative deepening loop

7. Improve the state evaluation function or implement a better one. Test if it is really better by pitching two agents (one with each evaluation function) against each other or by pitching each evaluation function against a random agent. If you run the experiments with the random agent, you need to repeat the experiment a decent number of times to get significant results. Don't forget to switch sides because typically one side has an advantage in the game. Run the experiments with several different board sizes and time constraints (play clock) between 1s and 10s. Report on those results and interpret them.

8. (**(optional, 10 bonus points)**) Implement transposition tables for reducing the number of revisited states. Note, that you can not simply discard revisited states, especially with α/β-pruning, it is non-trivial to decide whether a cached entry can be used or not. The wikipedia page on Negamax (`https://en.wikipedia.org/wiki/Negamax`) shows pseudo-code for Negamax with α/β-pruning and using a transposition table. More detailed information on transposition tables and implementation details for Chess can be found at `https://www.chessprogramming.org/Transposition_Table`.

   Evaluate the search with transposition table compared to the one without. How much reduction in search effort (number of state expansions / time) do you get?

9. Make your code fast and good! The more state expansions you get per second and the fewer states you need to expand for a particular depth limit, the better the player. Ideally, you should be able to solve the small boards (3x5, 5x5) in 10s (that is completely search the whole state space). There are two main ways to improve your code:

   (a) Make sure that state expansions are fast (part of this is also how many new objects are created during search, because the garbage collector will have to remove them at some point which takes time).

   (b) Improve alpha-beta pruning in order to reduce the number of states you have to search through for a particular depth limit. The main factor here is to order the moves well, that is, try to let the best move be most likely to be expanded first.

   When you try to improve your code, make sure to measure how much of an improvement you are making (e.g., time required to search to a particular depth and/or number of state expansions for a particular depth limit).

10. Write a short report containing:

    - description(s) of your state evaluation function(s)

Figure 1: *Game Controller Settings*

- a description of the experiments you ran (which games, which time constraints, how often, which agents in which roles, ...)
- the results of your experiments in a well readable form (tables, graphs, statistics; not just a list of matches and their results)
- your interpretation of the experimental results (What does that mean for the evaluation functions you compared, for the improvements you tried to make to your code, etc.?)
- a short description and evaluation of any improvements you have made to the program (move ordering, faster code, ...)
- Evaluation of the effects of using transposition table in case you implemented one.

Material

The code containing a Java and a python project for the lab can be found on Canvas.

The files in the archive are similar to those in the previous labs. The archive contains code for implementing an agent in the src directory. The agent is actually a server process which listens on some port and waits for a game simulator or a game playing robot to send a message. It will then reply with the next action the agent wants to execute.

The zip file also contains the description of the environment for different board sizes (`knightthrough_XxY.gdl`) and simulators (`chesslikesim.jar`, `gamecontroller-cli.jar` and `kiosk.jar`). To test your agent:

- Start the simulator (execute `chesslikesim.jar`).
- Setup the simulator as shown in Figure 1. The playclock setting determines how much time your agent has for each move.
- You can use your player as both the first and the second role of the game, just not at the same time. To let two instances of your agent play against each other, start your agent twice with different ports to listen on and use the respective ports in the simulator. You can change the port your agent listens on by given a port number as a command line argument when running the agent.
- If you use Java, run the “Main” class in the project. If you added your own agent class, make sure that it is used in the main method of Main.java. You can also execute “ant run” on the command line, if you have Ant installed. The output of the agent should say “NanoHTTPD is listening on port 4001”, which indicates that your agent is ready and waiting for messages to arrive on the specified port.
- If you use the Python code, run `gameplayer.py`. If you added your own agent class, make sure that it is used in the main method of`gameplayer.py`. The output should say something along the lines of “XYAgent is listening on port 4001 ...”, which indicates that your agent is ready and waiting for messages to arrive on the specified port.

- Now push the “Start” button in the simulator and your agent should get some messages and reply with the actions it wants to execute. At the end, the output of the simulator tells you how many points both players got: “Game over! results: 0 100”, the first number is for white and the second for black. The simulator will output scores 0 for loss, 50 for draw and 100 for win. These scores do not necessarily have to match the ones in your own state space model.
- You can also use the kiosk to play against your agent. The kiosk allows for human players to select moves, but it does not allow you to let two instances of your agent play against each other.
- To test your agents in multiple matches, the command line version of the simulator could be helpful (gamecontroller-cli.jar). For this to work, start your agent first before you run the simulator as follows:

```
# to let localhost:4001 play as white against a random player as black with
# a playclock of 5 seconds:
java -jar gamecontroller-cli.jar testmatch breakthrough_9x9.gdl 10 5 1 \
 -remote 1 Player1 localhost 4001 1 -random 2
# (all on one line)

# to let localhost:4001 play as white against a localhost:4002 as black:
java -jar gamecontroller-cli.jar testmatch breakthrough_5x5.gdl 10 10 1 \
 -remote 1 Player1 localhost 4001 1 -remote 2 Player2 localhost 4002 1
# (all on one line)
```

Hints

For implementing your agent:

- Add a new class that implements the `Agent` interface. Look at the `RandomAgent` class to see how this is done.
- You have to implement the methods `init` and `nextAction`. `init` will be called once at the start and should be used to initialize the agent. You will get the information, which role your agent is playing (white or black) and how much time the agent has for computing each move. `nextAction` gets the previous move as input and has to return the next action the agent is supposed to execute within the given time limit. `nextAction` is called for every step of the game. If it is not your players turn return `NOOP`.

  (In the Python version, these methods are called `start` and `next_action`.)
- **Make sure your agent is able to play both roles, white and black!**
- Do not try to implement everything at once. I suggest you do things in the order suggested in the tasks and test thoroughly after each step. You might want to even take this apart into smaller steps. E.g.,

  1. Implement the state-space model.

2. Implement an agent that keeps track of the current state of the game using the state-space model and is thus able to play legal moves. This can be tested by playing a number of matches of different game sizes against a random player. Make sure to also test playing the other role. In all cases the simulator should not print out any error messages.
3. Implement the state evaluation function and test it on a few hand-crafted states to make sure it is correct.
4. Implement a Minimax (or Negamax) search with a fixed depth limit to select the best move but do not implement pruning or iterative deepening yet. Test the search on a few small hand-crafted states where you know what should happen. It might also help to print out all the values that the search produces for intermediate states and re-construct the tree it goes through and the values it gets on paper to see whether it is correct.
5. Add alpha-beta pruning to the search and test it. Adding the pruning should not change which move is selected, only how many nodes need to be expanded to finish the search. Again test this properly.
6. Add iterative deepening to you search. Again, if the final depth-limit is the same as before, the same move and value should be returned.
7. Handle the time constraints properly.
8. Only now, try to improve the performance of the agent. Before you change something, think about how you can measure whether it actually improves the agent (increased number of node expansions/second, decreased number of nodes that need to be expanded until a certain depth, increased win rate against some other agent, etc.).

- Your agent should be able to play the small boards perfectly (at least the 3x5, probably also the 5x5), with a playclock of around 10 seconds. Playing perfectly, means that the move the agent chooses is independent of any evaluation of non-terminal states (i.e., the agent should not have to evaluate a non-terminal state with the state evaluation function in the last iteration of the search it does). In that case, alpha-beta search will find the optimal move. Essentially, this requires that the search is fast enough (and/or prunes enough branches) to expand the tree so deep that all leaf node are terminal states.

  You will need a decent heuristics to play the bigger boards well.

Handing in

You must hand in this assignment through Canvas. Hand in a PDF report and a ZIP archive with your code. The files in the ZIP archive must have the following structure:

```
project1
project1/build.xml
project1/python_src
project1/agent.py
...
project1/src
project1/src/Agent.java
...
```

Additional source files (like your agent class) should be added in the appropriate place.

Grading

- 75% - implementation (correct implementation of the model, algorithm and heuristics, quality/readability of the code)
- 25% - report

Bonus points:

- up to 5% for being fastest in solving the smaller environments (finishing the search with having expanded the tree all the way to terminal states)
- up to 5% for beating the other teams in the tournament
- up to 10% for correct implementation and evaluation of transposition tables