Programming Assignment 2 - Adversarial Search – Artificial Intelligence

Teacher: Stephan Schiffel

February 8, 2017

Use the Piazza page, 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

15 hours per student in addition to the time spend in the labs, assuming you work in groups of 2-3 students, did the previous programming assignment and and attended the lectures on search and adversarial search or worked through chapters 2, 3 and 5 in the book.

Problem Description

Implement an agent that is able to play the game of Breakthrough. This game is a simplified version of chess.

The game is played on an grid of width W ($2 \leq W \leq 10$) and height H ($4 \leq W \leq 10$). The initial state is setup such that both players have two rows of pawns of their respective color. White's paws are on rows 1 and 2 while black's pawns are on rows $H-1$ and H. The two players "white" and "black" take turns in moving one of their pawns. White moves first. Pawns move like in chess, that is, they can move one spot straight forward (up for white or down for black) onto an empty square or they can capture an opponents piece by moving one spot diagonally forward. As opposed to chess, pawns on the second row can not move two spaces at once. The goal of the game is to advance any one pawn to the opposite side of the board (i.e., to promote a pawn). The game ends, if one of the players has reached his goal or if the player who's turn it is does not have any legal move. This can happen if he does not have any pieces left, or if all his pawns are in positions where they cannot move. The game ends in a draw if none of the players wins. The scores are 100 points for winning, 50 points for a draw and 0 points for losing.

The legal moves for the agent are called `(move x1 y1 x2 y2)` meaning that the pawn 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 agent can play the game. Bonus points will be given for best agents.

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.
3. Implement a state evaluation function for the game. You can start with the following simple evaluation function for white (and use the negation for black): *50 - distance of most advanced white pawn to row H + distance of most advanced black pawn to row 1*
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, current depth limit of your iterative deepening loop and runtime 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.
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. Make your code fast and good! The more state expansions you get per second, 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). Improving the alpha-beta pruning could lead to being able to search deeper and thus get better results.
9. Write a short report containing:
    - description(s) of your state evaluation function(s)

- 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
- a short description of any improvements you have made to the program that you believe could be worth some bonus points (up to 5%)

Put the report as file `report.pdf` in the `answers` directory of the assignment on skel before handin in.

Material

The Java project for the lab can be found on Skel. See, instructions below on how to get it.

The files in the archive are similar to those in the first programming assignment. 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 (`breakthrough_XxY.gdl`) and simulators (`gamecontroller-*.jar` and `kiosk.jar`). To test your agent:

- Start the simulator (execute `gamecontroller-gui.jar`).
- Setup the simulator as shown in Figure 1. The playclock setting determines how much time you have 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.
- 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.
- 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.
- You can also use the kiosk to play against your agent. The kiosk does have visualization of the game, but it does not allow you to let two instances of the agent play against each other.

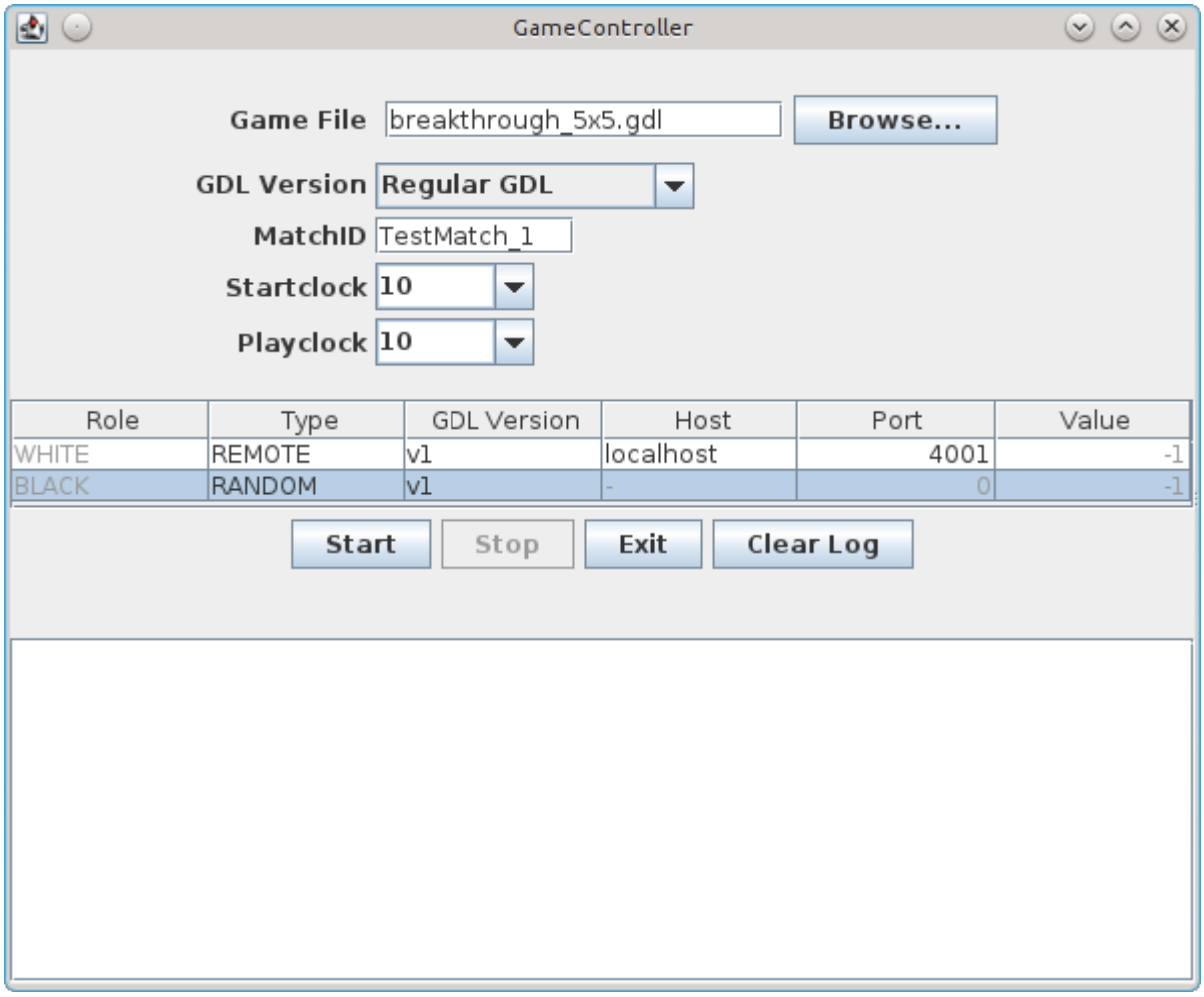

Figure 1: *Game Controller Settings*

- 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
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 `RandomAgent.java` 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`.
- **Make sure your agent is able to play both roles (white and black)!**
- You can make sure to be on time by regularly checking whether there is time left during the search process and stopping the search just before you run out of time, e.g., by throwing an exception that you catch where you call the search function the first time. Make sure you still return a sensible move (= one that was computed by a completed iteration of the search not by one that was stopped halfway through).
- 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. You will need a decent heuristics to play the bigger boards well.
- To specify the port your agent is running on change the build.xml file as follows and use the command line `ant -Dport=PORT run` with PORT being the port number:

```
...
<target name="run" depends="dist">
<java jar="${dist}/${projectname}.jar" fork="true">
<arg value="${port}"/> <!-- add this line here! -->
<jvmarg value="-Xmx500m" />
```

```
</java>
<antcall target="clean" />
</target>
...
```

Handing in

You must hand in this assignment through Skel (skel.ru.is) or it will **not** be graded. Only one member of the team should hand in the assignment. During submission you need to provide the RU usernames (skel logins) of all team members.

Connect to Skel using your favorite ssh client and unpack the assignment into your home directory by running the following commands:

```
[student14@skel ~]$ tar xvf /labs/arti17/prog2/prog2.tar
[student14@skel ~]$ cd arti17/prog2
[student14@skel prog2]$ ls
answers build.xml dist ...
```

You can copy the code to your own machine for development by using any SCP or SFTP client (e.g., WinSCP). However, you need to copy it back to skel into the same place before handing in. You can add more classes in the same package, but should not change the structure of the project. E.g., do not put the classes into a package other than the default package.

Before handing in, make sure you have put your report in the answers directory.

Finally, to handin your answers run `make handin` while being in the directory containing "Makefile". This should produce a file "/labs/arti17/.handin/prog2/student14/handin.tar.gz". You can check if it exists using the ls command.

Grading

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

Bonus points:

- up to 5% for beating the other teams in the tournament
- up to 5% for improvements to the algorithm that were not required (move ordering, transposition table, ...)