Artificial Intelligence

Game Description

344.021, 344.022, 344.023
WS2014

Contents

1 Updates/Amendments to this document

It will be the case that we have to update this document and/or the source code of the game, so before trying to work through the exercises, please make sure you have the newest version of both, after the exercise sheets are handed out.

2 The Game world

To complete the exercises in both the obligatory and the voluntary part, you will have to become familiar with the board, the game world to be precise, which is partly described in this document, and completely described by the code which implements the rules of the game. You will have to interact with this game world through the use of "intelligent", goal driven agents. The goals differ greatly for the obligatory and the voluntary part. Whereas the obligatory parts will have fairly easy goals which will usually look like "Find the (shortest) walkable path from point A to point B" and "Find the (shortest) sequence of actions to achieve the goal", the voluntary part only has a rather abstract goal, namely "win the game against your opponent".

The game is a turn-based version of a popular series of games, which has been around since 1983, with a few adaptations to make it easier to write AI agents for it. To avoid any legal liabilities regarding trademarks, we chose to rename all the game pieces. We could have chosen to make it completely abstract, but opted to borrow names associated with fables and 80ies pop-culture.

Should you encounter what you consider a case of an unclear or overly ambiguous description, the code implementing that part of the game is to be considered the ultimate reference on how things actually work. If you notice such oversights, please point them out to us, so we may improve upon the unclear description.

2.1 Board

The board that the game is played on, consists of two types of tiles: "Walls" and "Paths". As the name suggests, paths can be walked upon, and walls cannot. A typical board may look like one of the examples in Figure 2.1.

Figure 2.1: *Examples of typical boards*

The walls are rendered in dark gray, whereas the paths are rendered in a lighter shade.

2.2 Unicorns

The game pieces that the players (your programs!) will be controlling, are called unicorns, and for the two player game will come in two colors, red and blue (see Figure 2.2).

Figure 2.2: *An example of a board with two player-avatars (called "Unicorns"), one in blue, one in red*

2.3 Clouds

Clouds are pretty wet and unicorns do not want to go through them, so they can in fact be considered obstacles - they cannot be walked upon. See Figure 2.3 for the graphical representation. In this respect they behave exactly like walls. They can be removed though, with the help of the next few game pieces.

Figure 2.3: *An example of a board with a few clouds, in a very light gray*

2.4 Seeds

Each unicorn has the ability to spawn an unlimited amount of rainbow "seeds". A unicorn can spawn one seed per turn, at its current position, if there is no seed at this location already. This is the only time that a unicorn can be on top of a seed. Once it moves away from the location, it cannot move back there, as seeds block movement. See figure 2.4 for how they look in the graphic. After a few turns, the seed "sprouts" into a rainbow. These are helpful to remove obstacles, such as clouds. The seed takes 8 turns, until it sprouts.

Figure 2.4: *An example of a board with a seed shown in purple*

2.5 Rainbows

Rainbows sprout from clouds, in all directions. They extend exactly 3 tiles into each direction, counted from the originating seed. If there is a cloud in the way of the rainbow, it is briefly overlapped (for exactly 1 turn), and the cloud evaporates (is removed from the board). If there is a unicorn in the way of the rainbow, the unicorn cannot resist, and goes sailing in the skies. This effectively makes it leave the board, and therefore it automatically loses. In Figure 2.5 the graphical depiction of rainbows can be seen.

Figure 2.5: *An example of a board with rainbows sprouting from a seed. Rainbows are shown in light purple.*

2.6 Fountains

A fountain is a place on the board marking a special position. In the beginning, a fountain is unvisited. Each fountain remembers which unicorn it was visited by last. For each turn that passes, each fountain yields a point for the unicorn that has last visited this fountain. See Figure 2.6.

Figure 2.6: *An example of a board with two unicorns, and one "fountain". On the left, the fountain is shown in yellow, because it has not been touched so far. In the middle, the fountain has a blue core, because it has been visited by the blue unicorn last. On the right side, the fountain has a red core, because the red unicorn was the last to visit.*

2.7 Moves

In principle, there are 6 moves possible:

- STAY - the unicorn stays at its current location
- SPAWN - the unicorn stays at its current location, and drops a seed
- UP, DOWN, LEFT, RIGHT - unicorn moves one step in the direction indicated

Of course, not all of these are actually possible all of the time, as movement is primarily restricted by walls and clouds. Attempting an impossible move converts the impossible move to 'STAY'. After a move is executed, the board state is updated, and the tick of the board is increased by one.

3 Obligatory Part

For the obligatory part, you will have to implement various search algorithms as discussed in the lecture in the JAVA programming language. Your algorithms will be tested on a few problem instances, to determine whether they function according to the specifications. You will be provided with a set of problem instances for the different search algorithms in the form of unit tests. This is intended to help you verify that your implementations are working properly, before you submit them. Because unit tests can only be used to test a finite set of invariants, there is no real guarantee that even if you pass all the tests, your algorithms are correct or bug-free. Therefore you are of course encouraged to think of additional invariants to test for, and submit these too.

Once you submit your solutions, your implementations will be subjected to an additional set of problem instances unknown to you, to make sure there are no shenanigans of any sort. Also, because it's hard to test against our reference implementations, without actually giving you the solution.

3.1 Node Types and Example Implementation

Because we know that reading and understanding other people's code is hard and tedious, we will walk you through an example of how you could go about implementing a search algorithm. We will implement a (very) uninformed search, namely a random search. Before we can do that, we need to become familiar with the interfaces used for search. The package "`at.jku.cp.ai.search`" contains all the interfaces and classes we need.

To search any kind of space, we need to know how to get from one position (or state) to the next. There needs to be a neighbor-relation, or adjacency-relation, so we can tell where to go, given a specific state.

Given any kind of member of a state-space, be that a specific location on the board or a specific board itself, the "`Node`" interface specifies how we can obtain the adjacent locations or states in search space. It is held very simple, and declares a method named "`adjacent()`", which just returns all the nodes reachable in one step.

```java
public interface Node {
  <State> State getState();
  <Action> Action getAction();

  Node parent();
  List<Node> adjacent();

  boolean isRoot();
  boolean isLeaf();

  public int hashCode();
  public boolean equals(Object obj);
}
```

Lots of things have been done for you already - in the package "`at.jku.cp.ai.rau.nodes`" you can find several implementations of this interface. From here on, we will simply refer to these implementations as "nodes". Depending on the type of node, a node instance holds slightly different information. If you do not plan on competing in the voluntary part, feel free to skip the node descriptions. If you are interested in competing, you may want to know this, in case you decide to implement your own nodes.

3.1.1 IBoardNode

The "`IBoardNode`" implementation is used to search the **complete** state-space of a given board. If you call `adjacent()` on a node instance, a list of neighboring nodes in the state-space is returned, corresponding to **all legal** moves that the current unicorn can make. If all moves are actually possible, the moves `STAY, SPAWN, UP, DOWN, LEFT, RIGHT` are executed on copies of the current board, and the resulting new states are returned.

3.1.2 OnlySafeMoveNode

The "`OnlySafeMoveNode`" can be used to search within a **subspace** of the state-space of the board, which **does not** include the "dangerous" moves, like spawning seeds. This prevents your unicorn from endangering itself (not from your opponent!). If all moves are actually possible, the moves `STAY, UP, DOWN, LEFT, RIGHT` will be executed.

3.1.3 AlwaysMoveNode

The "`AlwaysMoveNode`" does not `STAY`. It always tries to move a unicorn. If all moves are actually possible, the moves `UP, DOWN, LEFT, RIGHT` will be executed.

3.1.4 OnlyPositionNode

The "`OnlyPositionNode`" governs search in the state-space given by the **passable areas** of a concrete board. It can be used to compute paths **without** taking **any other** changing state into account. This may be useful for making distance calculations on a specific board. This

node **does not know anything about unicorns, seeds or rainbows**! The only thing it cares about is **walkable** tiles on **the board it was given** to begin with.

3.1.5 Random Search

We have provided a skeleton class for each part of each exercise, and we'll now look at the skeleton for random search in this example. It will more or less look something like this:

```
1  package at.jku.cp.ai.search.algorithms;
2
3  import java.util.function.Predicate;
4
5  import at.jku.cp.ai.search.Node;
6  import at.jku.cp.ai.search.Search;
7
8  public class RS implements Search
9  {
10   @Override
11   public Node search(Node start, Predicate<Node> endPredicate)
12   {
13     // TODO
14     return null;
15   }
16 }
```

We note that there is one method that needs to be implemented, namely the method "`search`". The first parameter we get is some type that implements "`Node`", which is great, because then we have some node to start from. The second parameter is a predicate on nodes, which will tell us when we have found our goal. How do we proceed from here?

Well, a few points to consider (in no particular order) would be: consult the literature for "random search", start hacking away at the keyboard, look at the unit tests - are they all green yet?, continue hacking away at the keyboard, pause to think for a while, discuss the problem with your partner, go for a walk, think about differences between search in trees and graphs.

We'll fast-forward here, and just take a look at the file `at.jku.cp.ai.search.algorithms.RS`. The implementation you got provided still has a small bug which you'll have to fix, in order to complete assignment 0.

4 Search in graphs

4.1 Cycle Avoidance

We are searching in a very loopy state-space. We need to remember where we have been, otherwise we'll go there again. There are three possibilities:

a) don't visit parent nodes again

b) don't create cycles (don't visit nodes on the current path again)

c) don't visit any node we have visited before (keeping a "closed list")

Option “a” is the least costly in terms of space, because we have to remember the parent of a node in any case - on the other hand it is the least useful, in terms of work-avoidance, because it does not prevent us from having cycles with a length greater than one.

Option “b” is the middle ground in terms of space. We have to remember as many nodes as the length of the path we are currently on. It prevents us from cycling.

Option “c” is the most expensive, in terms of space, as we may need to keep around on the order of $O(b^d)$ nodes.

4.2 Self-Avoiding DLDFS in graphs

Properly implemented DLDFS(d) has a runtime complexity of $O(b^d)$, and a space complexity of $O(bd)$, if we **do not keep** a “closed list”.

DLDFS expands $O(b^d)$ nodes. If we would keep all of them in a “closed list”, we’d increase its space-complexity to $O(b^d)$ nodes too.

To keep the lower space-complexity, we can only remember the **current** path we are exploring. Therefore, the best the algorithm can do to not repeatedly expand nodes, is to avoid expanding nodes which are contained **in** the current path. This is sometimes also called a “self-avoiding walk”, because we avoid stepping where we already have been.

DLDFS is usually implemented with a stack. Either recursively and implicitly via the call-stack, or iteratively and explicitly via a separate stack. In order to check whether a newly expanded node was already encountered, we need a stack that has a fast “`contains`” operation, which incidentally could be found in the class “`StackWithFastContains`”. This data structure essentially keeps a “`Set`” parallel to the “`Stack`”.

4.3 Priority Queues

A priority queue is a data structure which stores elements along with an associated priority. This means that it represents an ordered sequence of pairs (p, e). The two interesting operations that a priority queue has to offer are “`insert((p, e))`” and “`retrieve()`”. You can insert the elements in any order you like, the “`retrieve()`” operation guarantees that you will always be given the pair with the highest priority in $O(1)$ runtime.

Because priority queues are usually implemented as MIN-HEAPS or MAX-HEAPS (depending on what “priority” means to you), they are not “stable” when there is a tie between two equal priorities. This means that if there are two pairs (p, A) and (p, B), and p is the highest priority right now, *it is unspecified, which of the two pairs* the “`retrieve()`” operation will return.

To make your implementation’s solutions comparable to ours, we had to make sure that the order, in which “`retrieve()`” returns prioritized elements in case of a tie, is deterministic. We therefore provided a “`StablePriorityQueue`” class which falls back on the insertion order, in case of a tie. Incidentally, this makes it usable as a standard FIFO data structure (also called “queue” ...), but causes a lot of unnecessary comparisons, to determine retrieval order. So please use a regular queue where it is appropriate.

5 Troubleshooting, Debugging, Visualization, Who To Blame

If you get stuck at any point during implementation, because your implementations do weird things, or you are convinced that you did the right thing, but the unit-tests are still failing, it could be a good idea to visualize what your algorithm is doing. All the game-objects, as well as the board are printable, so a simple "`System.out.println(node)`" should get you started. Be aware that objects may be in the same location on the board together, when looking at the board representation.

For a more complex visualization of the search-space, there are some routines provided in the class "`RenderUtils`", as well as some usage in the class "`VisualizeBoards`", in the "`utils`" and "`visualization`" sub-packages. These were added as a mere afterthought, born from a desire to actually see what is going on, and we decided to leave them in.

If you are still convinced that your implementations are correct, and the unit-tests are wrong, please contact us. Ideally you'd include proof that it is really our implementation that is at fault, such as a minimal test case that fails.

We are far from perfect, and accidents and bugs may (have) happen(ed). The person that is most likely responsible for any blame in this case is Rainer. You can find his contact information at the end of this document. Please e-mail only.

6 Voluntary Part - The Game Rules

The game is of the type turn-based, zero-sum, with perfect information, played by two players. That sounds overly complicated, so let us discuss each point in turn:

- "Turn-based" means that players take alternating turns in making their moves.
- "Zero-sum" means that the game is strictly competitive, and the gains of one player are the losses of the other (and they usually sum to zero, hence the name), which means that there can be only one winner. However, there is also the possibility of a "draw", meaning both players are equally winning and losing simultaneously.
- "perfect information" means that at every turn, all information regarding the current state of the game (board, game pieces, possible actions) is accessible to the player.

6.1 Rules

The game will be played by two players (your programs!), taking turns. To plan the next move of your player, your program will have to "think" (or search, in our case). Thinking time and space is limited. We limit the thinking time to 5 minutes for each player, for the whole game. The space limit for storing/remembering is fixed to 512M. The game takes place on a board with fixed walls, some removable clouds, and some fountains. The objective of the game is to win. The winner is determined at the end of the game. The game ends when any one of the following conditions is met at the end of a turn:

- The turn-limit is reached (exact number may vary)
- A player's avatar has been taken off the board
- The time has run out for a player during thinking
- The space has run out for a player during thinking

The winner is determined as follows:

- If there is only one player left on the board, this player wins
- The player with the higher score wins, once the turn-limit is reached
- If the time has run out during thinking, the other player wins
- If the space has run out during thinking, the other player wins

Until the competition actually starts, you are free to experiment with the game framework. There will be more information and documentation on how to run a game with two agents. In the meantime, see the class "`at.jku.cp.ai.competition.runtime.Runtime`". This class enables you to test your players before you upload them.

7 Competition

In order to participate in the competition, your group will have to upload an agent (in MOODLE), contained in a **zip** file, named after your group. If your groupname is "`g00`", the file should be called "`g00.zip`". You should not include compiled classes, only the source code, and with the same directory layout as a submission for an assignment. See assignment sheet zero for (way too much) details.

7.1 Source Code

The code for your bot, as well as any supporting code must go into a package named:

"`at.jku.cp.ai.competition.players.<groupname>`"

For example, if your groupname is "`g00`", you create a package

"`at.jku.cp.ai.competition.players.g00`"

The main bot-code will go into a class implementing the "`Player`" interface. It shall be located in a class also named after your group. If your groupname is "`g00`", the class implementing the interface would be named:

"`at.jku.cp.ai.competition.players.g00.G00`"

Every time it is your bots turn, its "`getNextMove`" method will be called with the current game state (time left, etc...) and the current board state. This method should return the next move your bot wants to play.

Every dependency your bot has on any other (helper) classes, or libraries you wrote, has to go into the player package! This **includes every** search algorithm, **every** subtype of `Node` you might have come up with, etc. You are at liberty to do what you want below this directory, in terms of organization, as long as you adhere to the naming-scheme for the player class, you will be fine.

7.2 Environmental Issues and Code of Conduct

Because we are basically running untrusted code, we have taken some precautions that this code does not do anything weird to your opponents code, or to the machine its running on. Your bot code will run within its own JVM, and communicate with the game server. All of the communication with the server is handled by the framework, you don't have to do anything extra there.

Particularly: don't try to open any sockets, don't try to open any files, don't try to poke around the system. This is neither a wargame nor a pwn2own contest.

Any attempt to disrupt communication with the game server or any attempt to leave your sandboxed environment, will lead to your immediate disqualification, (very) angry sysadmins and may even lead to all of your group members **failing** the course.

Also, please keep the amount of "`System.out`" calls to a minimum, ideally **0**.

7.3 Example Bots

In order to get you started with bot development, two commented example implementations are provided in the class

"`at.jku.cp.ai.competition.players.example.Example`"

and

"`at.jku.cp.ai.competition.players.headless.Headless`"

The "`Example`" bot is only going to work with an implemented AlphaBetaSearch, and the "`Headless`" bot needs a working BFS implementation. These two bots provide a start for bot development, they are not really competitive - actually they are both pretty bad. The "`Example`" bot uses an adversarial search, combined with a scoring function (which is bad), and the "`Headless`" bot's behaviour is modelled using a purely reactive state machine (which does not look ahead in time, hence the name).

7.4 Running a match

For testing purposes, you can run your bots within the same JVM, via the `Runtime` class. You call it like this:

```
java at.jku.cp.rau.runtime.Runtime \
    <p0> <p1> <level> \
    <timelimit [s]> <movelimit> <seed> <[verbose|silent]>
```

The following commandline would tell the `Runtime` to load two instances of the `Headless` player, load the arena called `arena.lvl` with a time limit of 300 seconds, a move limit of 100 and a random seed of 42, and be `verbose` about what it is doing. (use `silent` if you are only interested in the outcome of the match)

```
java at.jku.cp.ai.competition.runtime.Runtime \
    at.jku.cp.ai.competition.players.headless.Headless \
    at.jku.cp.ai.competition.players.headless.Headless \
    assets/arena.lvl \
    300 \
    100 \
    42 \
    verbose
```

7.5 Tips

To make your bot competive, you'll have to think about a proper scoring function that judges whether certain board positions are better or worse than others. Here is an (incomplete) list of things you might want to think about when constructing your scoring function:

- how dangerous is it currently? (could we lose in a few moves?)
- how far are we from our opponent?

- could we place a seed to win the game?
- could we lose if we place a seed now?
- could we place a seed to slow our opponent down?
- how many fountains did we color yet?
- can we remove clouds to get to a fountain?
- are we in a dead end - could we get imprisoned by our opponent?
- if we are behind in points, can we force a draw?

8 Setup

We provide the setup-instructions for the eclipse - IDE:

- obtain JAVA SE 8 or greater - download the JDK at "`www.oracle.com`"
- download eclipse at "`www.eclipse.org`"
- download the game framework archive from the MOODLE course website
- unpack the archive - you will be left with a folder "`aiws15`"
- start eclipse, point the workspace to the directory you unpacked "`aiws15`" into
- create a new java project, name it "`aiws15`", click "`finish`", and the wizard will do the rest
- make sure the JAVA compiler and runtime environment are actually set to JAVA SE 8, by right-clicking the new project, selecting "`Properties`", and looking at the "`Java Compiler`" settings. The "`Compiler Compliance Level`" as well as the "`Source Compatibility`" should say "`1.8`"
- right-click the new project, go to the menu point "`Build Path`", then to "`Configure Build Path...`", then select "`Add Library`" and there you select "`JUnit`" (JUnit 4 to be precise).
- (only if necessary - sometimes the wizard fails to include these) Select "`Add JARs`" next, locate the `libs` folder, and add the `equalsverifier-1.7.5.jar` file, as well as the `prefuse.jar` file, if you want the (experimental) visualization.
- your project should be setup by now - let's test that
- right-click the project again, and select "`Run As`" and then "`JUnit test`"
- all the unit-tests within the project should be executed now

- the unit tests should all be green, **except** the ones in the "`tests.assignment<i>`" packages of course
- if you want to skip all other unit-tests, repeat the procedure on the "`tests.assignment<i>`" package only

9 Contact

For any questions and/or discussion of results contact

Rainer Kelz
rainer.kelz@jku.at
Room SP3 444, Science Park 3, 4th floor

Filip Korzeniowski
filip.korzeniowski@jku.at
Room SP3 444, Science Park 3, 4th floor