The Fruit Rage!

(a homework and a programming contest)

CSCI-561, Spring 2012 / Professor Laurent Itti

Homework II

Developed by Nader Noori

Due date : 03-05-2012

Introduction

This project will provide you with the opportunity to practice what you learned about the game playing and the adversarial search in the class.

So far, you have learned how to use different adversarial search algorithms in playing a zero sum game. Zero sum games are indeed an abstraction of those situations in which agents compete for some fixed common resources in a turn taking fashion. In a typical zero sum game two player game, each player's gain is measured by their share of a common resource and the goal is maximizing the gain.

Often agents in a realistic environment have to satisfy other constraints on other personal resources such as time, energy or computational power in the course of the game playing. In this homework we will introduce '**The Fruit Rage!**', a game that captures the nature of a zero sum two player game with strict limitation on allocated time for reasoning.

Your task is creating a software agent that can play this game against another agent. You will be provided with a game engine, an interface for implementing your agent and two implementations of this interface. The provided interface includes a function (method) which is called by the game engine when it is your turn for move. Your agent should compete with another player for a shared resource (fruits in a box), while using a limited amount of CPU time.

Rules of the game

The Fruit Rage is a two player game in which each player tries to maximize his/her share from a batch of fruits randomly placed in a box. The box is divided into cells and each cell is either empty or filled with a fruit of a specific type. In the beginning of each game, all cells are filled with fruits.

Players play their turn in consecutively. Players pick a cell of the box in their own turn and claim all fruit of the same type in all cells that are connected to selected cell through horizontal and vertical paths. For each selection or move the agent is rewarded a numeric value which is the squared value of the number of fruits claimed in that move. The total gain during the course of game playing is simply the sum of movement gains. Once an agent picks the fruits from the cells, their empty place will be filled with other fruits on top of them.

Another big constraint of this game is that every agent has a strict limitation on their total CPU time used during the course of the game playing. Using a total CPU time beyond the limit will be punished harshly.

The overall score of each player is the sum of rewards gained for each of turns (total gain) minus the penalty value. The game will terminate when there is no fruit left in the box.

The following figures depict the first two iterations of the game, set in a 10x10 box of fruits with 4 different types of fruits denoted by 0,1,2 and 3. The first image shows the initial state of the fruit box which is set randomly. The first player decides to pick the cell which is marked with red font color in a yellow background. In the next figure all the fruits connected to the selected cell which have the same type of fruit are taken out from the box, and their empty place is marked with * s. The first player will claim 14 fruits of type 0 because of this move and thus will be rewarded 196 points.

Figure 3. shows the state of the game after the empty space is filled with fruits falling from top cells. In this image the choice of the second player is marked in green font with a yellow background. Upon selecting this cell all 12 fruits of type 1 connected to that cell will be given to the second player and thus the second player will gain 144 points. In figure 4, cells connected to the selected cell are marked with * and in figure 5 you see those picked fruits are replaced with the content of top cells.

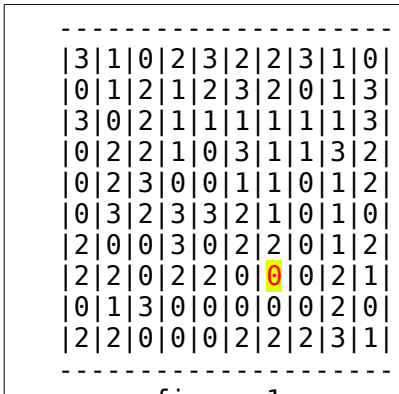

figure 1.

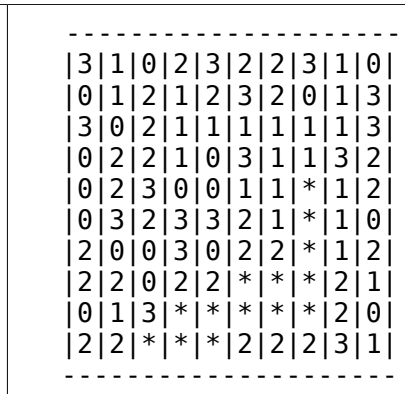

figure 2.

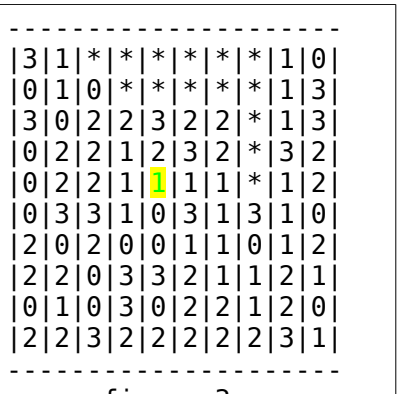

figure 3.

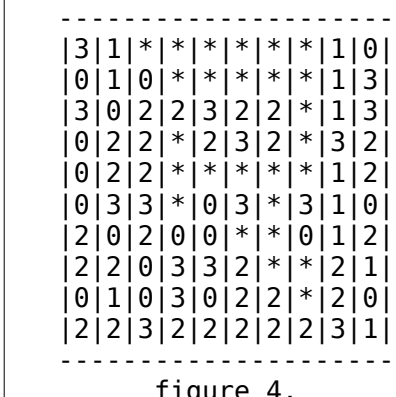

figure 4.

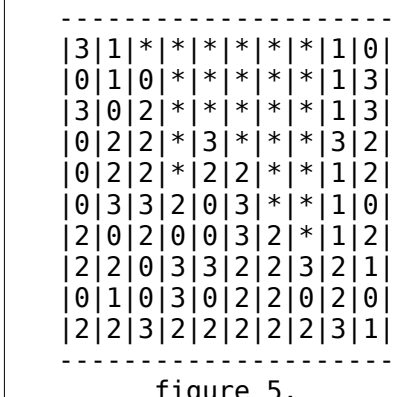

figure 5.

Thus, every move will change the layout of the game board. Sometimes a move which seems very good for a greedy player will open the door for the opponent to score even a higher point. The challenge is selecting those moves that will yield a good point for the player without providing a better opportunity for the opponent to achieve even a better score. So being able to look farther ahead is crucial for a rational game player. But looking farther requires more thinking (which translates to more time if you have to use a certain hardware). In a realistic environment the whole world won't stop for you to think enough and thus you have to decide to cut off the reasoning chain at some stage and act based upon your best of knowledge. In this condition a smarter solution is the one that helps optimizing your resources to look further and use your time wisely. So a smarter agent is the one who knows where to look for a better move and meanwhile has an optimized implementation of its algorithm.

Your task

1. The Programming part (up to 100%)
Your task is writing a software agent in Java programming language, that is able to play smarter than a greedy agent. In addition to a game engine and other supplementary classes, you will be provided with the implementation of two agents : GreedyPlayer and RandomPlayer.

RandomPlayer is an agent who randomly finds a non empty cell of the game board for its next move It's almost impossible to lose to this agent unless you use your CPU cycles for some nonsense and non-relevant reasoning. So you will receive 0% for losing to such a dumb agent.

GreedyPlayer is a player that looks only one step ahead for the best move if not out of time and will perform like a random player when it runs out of time. In order to assess your agent you should play against this agent and make sure that you can beat this agent safely. For a solution that beats only RandomPlayer and loses to this agent you will receive only 20%. Winning **strongly** against GreedyPlayer will secure 70% of the total points of this assignment. If you win with a small margin you'll gain as low is 40%.

We will also hold a tournament between all agents written by students in the class plus GreedyPlayer and RandomPlayer. The agent which ranks the 1st will bring 30% extra credit for its programmer. Agents who stand in the 2nd and the 3rd position will receive 20% and 10% extra credit for their achievement, subject to rank above GreedyPlayer.

Size of the game board for the competition will be 32x32 with 5 different type of fruits. You'll be given 10 seconds of CPU time for each round of the game.

In terms of implementation you need to implement a single public method which is called by the GameEngine class. This method is introduced in IPlayer Java interface. For the sake of time keeping and providing some basic information to your agent we already have implemented some methods in an abstract class named Player that implements IPlayer, **so your start point will be extending Player class and implementing its only abstract method**.

All the classes are part of a package named `edu.usc.csci561.project2` and you need to implement your class within this package. Your implementation should be compatible with Java 1.6 and you are free to use all packages provided in Java SDK 1.6.

How to run the game engine:

First you need to compile the Java code, if you read this document you have already unpacked the prj2.tar.gz using this command:
gunzip prj2.tar.gz
tar -xvf prj2.tar

Having it done so, you've created this directory in your working path : `./csci561-project2/src`. From now on we call this directory as the source directory. Change your active directory to the source directory and issue this command:

```
javac edu/usc/csci561/project2/*.java
```

Doing so you will create your class files in `./edu/usc/csci561/project2/`, you can repeat this step whenever you change something in your code.

To run the game engine stay in source directory and issue the following command :

```
java edu.usc.csci561.project2.GameEngine edu.usc.csci561.project2.{Player1}
edu.usc.csci561.project2.{Player2}
```

where `{Player1}` and `{Player2}` should be replace by the name of agent classes that have extended `edu.usc.csci561.project2.Player`, an additional optional argument will let you see to choose to see the course of problem solving. Passing o (lower case o) as the third argument will generate the output after everybody's move while O (capital o) will also show the state of the game board right after choosing the move and before the gaps are filled with fruits of top rows. Seeing the output will not take your CPU time thus it will not interfere with the course of problem solving.

In your implementation YOU ARE NOT ALLOWED TO CALL OTHER AGENTS FOR ANY REASON, EVEN GREEDY OR RANDOM PLAYERS, VOILATING THIS CONDITION BY CALLING OTHER AGENTS IN ANY FORM IS TREATED AS CHEATING IN YOUR HOMEWORK. So keep in mind that you are not allowed to call even GreedyPlayer or RandomPlayer within your code, but you are allowed to use the source code of only these two agents by re-implementing, or copying parts of their implementation in your code.

Although hacking all parts of the code in the supplementary material is okay, but you need to make sure that you keep an intact copy of the code for your final testings.

For the coding part you'll submit only your player agent Java file in the form of _(R#)Player.java where R# is your roster number.

2.The written Part up to 40%

In addition to programming part you need to hand in you answers to the following questions on the due date of this homework in the class.

a. Explain your approach for computing your move. (10%)

b. Discover the boundaries of problem state space that your solution can not beat the greedy algorithm, you need to support your argument with some data gathered by testing your player against the greedy player on problems with different sizes and different time limits. (10%)

c. Discover the boundaries of problem state space that your solution can not go beyond a random algorithm. You need to support your argument with some concrete data. (10%)
(10% extra credit for a very nice and funky study of the behavior of your solution).

Good Luck!

N.N.