CSCI 561: Foundations of Artificial Intelligence
Instructor: Prof. Laurent Itti
Homework #2: Reversi Competition
Due on Oct 20 at 11:59pm, 2014

Introduction

This competition is optional and voluntary. It will not affect your grade in CSCI-561. We will run participating students' algorithms against each other in a tournament (like in tennis). The winner will receive a prize (to be arranged) and an opportunity to describe their algorithm to the class. The finalists, semi-finalists, and quarter-finalists will also be recognized (to be arranged).

If the task number is 4, your program will enter the competition mode. In this mode, you are free to use any algorithm, evaluation function, and depth limit to compete with other agents. The only constraint of this game is that every agent has a strict limitation on their total CPU time used during an entire game (sum over all moves until game over). If your agent uses a total CPU time beyond the limit, the game server will assume a random legal move for you until game over.

Task 4

In task 4, input.txt will contain:

```
<task = 4 for competition>
<your player: character X or O>
<your CPU time left in seconds: floating point number>
<state description, same as in HW2>
```

and output.txt should only contain:

```
<your next move>
```

where a move is a string with 2 characters: column name (a to h) and row number (1 to 8). For example, a6, c4, h5.

Makefile targets:

agent: (required) compiles the agent (and possibly some calibration code; see below) if needed. Any error during "make agent" eliminates your agent from the competition.

calibrate: (optional) This will be called once after "make agent" succeeds. You can use this to run some speed calibration program. You can write some file in the local directory that contains

the resulting calibration data for future use by your agent program. Any filename is ok except those already used (input.txt, Makefile, output.txt, etc). Errors during "make calibrate" are ignored and do not withdraw your agent from the competition.

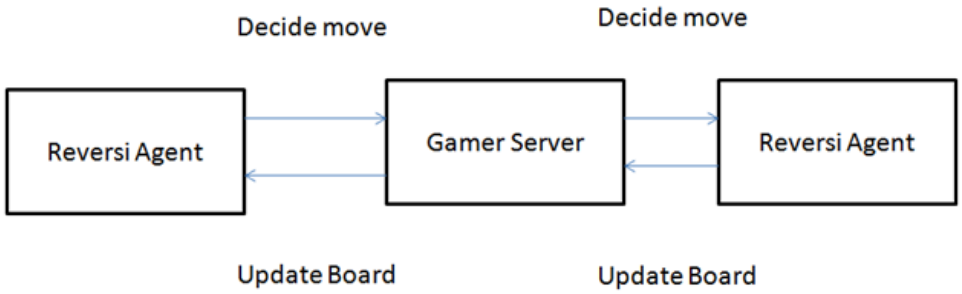
newgame: (optional) is run once before a new game. You can use this target to delete any files you might have stored on disk during a game (persistent information across moves; see below for explanations). Errors during "make newgame" are ignored and do not withdraw your agent from the competition.

run: (required) is called once per move. Your program should read input.txt in the local directory and write output.txt. Any error during "make run" is ignored, but, if one is detected, a random move will be chosen automatically for your agent.

Game playing workflow

The competition will be a tournament (like in tennis): We will make random pairs of agents, and play the 2 agents in each pair against each other several times. The winner of a majority of the several games in each pair advances to the next round (i.e., will play against another randomly selected agent that also won in the current round), while the loser is eliminated.

The workflow during the competition will be as follows:



Each student's agent will be in a directory with the student name. All files should be written in this directory. This will be set as the current directory during play. So, when you read or write files, do not specify any directory name, only the file name (e.g., "input.txt").

Before the competition:

- We will loop over all student directories; assume the current one is called "student"
    - cd student
    - make agent. If fails, withdraw student from competition.
    - make calibrate. Ignore any errors. If does not complete after 5 minutes of elapsed (wall clock) time, kill program at wall clock time = 5 minutes.
    - cd ..

Among all remaining students, select random pairs that will play against each other.

Competition:

- We will loop over all remaining student pairs. Assume the two selected students for a given pair are called "student1" and "student2"
    - loop N times (games) (N to be determined later; it will be an odd number; likely N=5):
        - randomly assign roles, eg, student1 is X and student2 is O
        - create initial state (note: likely will be the standard initialization as shown in the first figure of HW2 text, but this is not guaranteed; it is guaranteed that at least one legal move will exist for your agent)
        - initialize 2 time counters (one for each agent) to M seconds (see below).
        - cd student1; make newgame; cd ..
        - cd student2; make newgame; cd ..
        - on even values of N, student1 plays first, on odd values, student2 plays first, e.g., let's assume student1 plays now:
        - loop until game over:
            - cd student1
            - copy current state to input.txt; patch line 2 of input.txt to indicate correct player (here, X for student 1); patch line 3 to indicate remaining time for student1 in seconds.
            - check if game over and, if so, determine winner and go to next game.
            - check if no move exists for student1, and, if so, pass and go to student2.
            - if remaining time for student1 <= 0.0, select a random move.
            - else run "time make run" and capture the CPU time value. If program did not complete after H seconds wall clock time (hard limit per move), kill agent and student1 loses
            - subtract CPU time used on this move from time counter for student1
            - read output.txt and update state using student1's move. If move is illegal, student1 loses.

            - cd ../student2
            - copy current state to input.txt; patch line 2 of input.txt to indicate correct player (here, O for student 2); patch line 3 to indicate remaining time for student2 in seconds.
            - check if game over and, if so, determine winner and go to next game.
            - check if no move exists for student2, and, if so, pass and go to student1.
            - if remaining time for student2 <= 0.0, select a random move
            - else run "time make run" and capture the CPU time value. If program did

not complete after H seconds wall clock time (hard limit per move), kill agent and student2 loses
- subtract CPU time used on this move from time counter for student2
- read output.txt and update state using student2's move. If move is illegal, student2 loses.
- check if game over and if so, determine winner and go to next game.

- After all N games are complete between student1 and student2, declare winner (student1 or student2) as the one who won the most games.
- winner advances to the next round of the competition, loser is eliminated.

Values:

These will be tuned as we receive feedback from students. Initial guess is:

- N = 5 (number of games at each round between two opponents)
- M = 200 seconds of total play for each player during one full game
- H = M (remaining) + 10 seconds grace period to account for I/O

Notes:

1) State in input.txt is guaranteed valid. In particular, because we check for game over and "pass" moves externally, you are guaranteed that at least one legal move remains for your agent when "make run" is called. You do not need to worry about cases where no move is possible for your agent.
2) There is one important difference between task 4 (competition) and the other 3 tasks of HW2: in task 4, the evaluation function is entirely up to you, and the rules for game over now follow the "official" reversi rules: when no more valid move remains, game is over and we count the number of X and O pieces on the board; the player with the larger number of pieces wins the game (+1 for the winner, 0 for the loser; it does not matter by how many pieces one won, e.g., 26 X, 21 O -> X wins +1; 2 X, 60 O -> O wins +1). If the numbers of X and O pieces are the same, we will use CPU time remaining to break ties. After N games, the winner is the agent that won the most games (no ties here since N will be an odd number).
3) Sorry we cannot give you the game server (given a state and a move, it computes the resulting state). You may want to implement your own if you want to test different versions of your agent playing against each other. It should not be difficult, since most of the code you need to write the game server should already be in the code for your agent (that is why we will not give it to you).

4) The time counters are for total playing time over one entire game. You should think of strategies to best use this time throughout the game. For example, do you want to spend more time early in the game if that means less time remaining to play later in the game? Or the opposite?
5) Calibration: "make calibration" is provided for you to estimate how fast the computer is that will run your agent. You can save a file in the local directory with any calibration data that your agent can then load later. It is called only once, just after compiling your code. It is optional. One might, for example, run a fixed game stored in the program and that is known to expand some number of nodes, and measure the time spent.
6) Persistence between agent calls: you are allowed to write files in your current directory, to possibly maintain some persistent information across successive calls to "make run". Usually, you would delete these files in "make newgame" unless you want to keep some of them from one game to another. For marshalling / serialization of your persistent data to disk, we encourage you to use libraries, such as *boost serialization,* or the *cereal* library written by prof Itti's students:
http://www.boost.org/doc/libs/1_54_0/libs/serialization/doc/index.html
http://uscilab.github.io/cereal/
7) Time used is the "user" time as returned by the Unix "time" command. This only counts time spent by your algorithm and discards time spent in file I/O or other system calls. Note, however, that the hard time limit is based on the "real" (or wall clock) time elapsed since we do not have access to your CPU time usage while your program is still running.
8) Multi-threading is allowed but note that CPU time measured is the sum of all CPU time over all threads (e.g., if you run in 5 seconds with constant 478% CPU load, then your time used is 5 x 4.78 = 23.9 seconds). Hence, multi-threading is not encouraged, it will not buy you any extra time.
9) If you need to know time elapsed so far during a run of your agent, we encourage you to use the C++11 chrono facility, see, e.g., http://en.cppreference.com/w/cpp/chrono . Typically, you would get an initial time point from the high_resolution_clock when you start work, and you can then get time points and compute durations from that initial time point later during execution of your algorithm. Note that this will get you access to elapsed (wall clock) rather than CPU time, which should be ok for single-threaded programs (recommended). You may also want to check out the getrusage() system call (see "man getrusage") if you need more precise access to your CPU time, but keep in mind that the hard limit after which your agent is killed (H) is based on wall clock time anyway.
10) Note that there is a bit of slack between H and M (10 seconds grace period). We recommend that you do not try to exploit this (i.e., aim to use M seconds of CPU exactly, not M+10). We may impose penalties for any time used beyond M, possibly to break ties among agents.

Run environment:

We will run the competition on the cluster in prof Itti's lab. Specs are as follows:

- Each node (computer) is a quad 12-core Opteron (48 cores total) 2.5GHz, 128GB or 256GB of RAM. Note that we will run 48 games in parallel on each machine. Again, multi-threading is not encouraged (it will not buy you any CPU time). Here are the full specs if you care about them:

```
itti@n12:~$ cat /proc/cpuinfo
processor : 0
vendor_id : AuthenticAMD
cpu family      : 16
model           : 9
model name      : AMD Opteron(tm) Processor 6180 SE
stepping  : 1
microcode : 0x10000d9
cpu MHz         : 2500.000
cache size      : 512 KB
physical id     : 0
siblings  : 12
core id         : 0
cpu cores : 12
apicid          : 16
initial apicid : 0
fpu       : yes
fpu_exception   : yes
cpuid level     : 5
wp        : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic
sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2
ht syscall nx mmxext fxsr_opt pdpe1gb rdtscp lm 3dnowext
3dnow constant_tsc rep_good nopl nonstop_tsc extd_apicid
amd_dcm pni monitor cx16 popcnt lahf_lm cmp_legacy svm
extapic cr8_legacy abm sse4a misalignsse 3dnowprefetch osvw
ibs skinit wdt nodeid_msr hw_pstate npt lbrv svm_lock
nrip_save pausefilter
bogomips  : 5000.07
TLB size  : 1024 4K pages
clflush size    : 64
cache_alignment     : 64
address sizes  : 48 bits physical, 48 bits virtual
power management: ts ttp tm stc 100mhzsteps hwpstate

[repeated 48 times]
```

- Linux Ubuntu 13.10 x86_64
- Your agent will run as a “regular” user (no root privileges).
- We will run your agent chroot’ed into a limited environment (i.e., you will not be able to read/write any files outside your agent’s directory; trying to open a file outside your directory will be blocked by the system and could crash and eliminate your agent).
- Stock Ubuntu 13.10 packages are installed (e.g., g++4.8.1, gcj-4.8.1, python-2.7.5+, boost-1.54).
- Custom packages will be considered only if there exists an Ubuntu 13.10 x86_64 package for it, so that we can install them by typing “apt-get install something” (i.e., we will not compile packages from source, etc)