Assignment 4

Markov Decision Processes

Deadline: November 30, 11:59pm.
Points for undergraduates: 120. Points for graduate students: 110. Perfect score: 100.

Assignment Instructions:

Teams: Assignments should be completed by teams of students - three members for undergraduates and two for graduate students. Mixed teams between undergraduates and graduates are discouraged. If formed, they will be graded as graduate teams. No additional credit will be given for students that complete an assignment with fewer members than the recommended. Please inform the TAs as soon as possible about the members of your team so they can update the scoring spreadsheet (find the TAs' contact info under the course's website: `http://www.pracsyslab.org/cs440`).

Submission Rules: Submit your reports electronically as a PDF document through Sakai (`sakai.rutgers.edu`). For programming questions, you need to also submit a compressed file via Sakai, which contains your code. Do not submit Word documents, raw text, or hard-copies etc. Make sure to generate and submit a PDF instead. Each team of students should submit only a single copy of their solutions and indicate all team members on their submission. Failure to follow these rules will result in lower grade in the assignment.

Program Demonstrations: You will need to demonstrate your program to the TAs on a date after the deadline. The schedule will be coordinated by the TAs. During the demonstration you have to use the file submitted on Sakai and execute it either on a university machine at CORE/HILL center (one of the CS labs) or on your laptop computer. You will also be asked to describe the architecture of your implementation and key algorithmic aspects of the project. You need to make sure that you are able to complete the demonstration and answer the TAs' questions within the allotted 12 minutes of time for each team. If your program is not directly running on the computer you are using and you have to spend time to configure your computer, this counts against your allotted time.

Late Submissions: No late submission is allowed. 0 points for late assignments.

Extra Credit for LaTeX: You will receive 10% extra credit points if you submit your answers as a typeset PDF (using LaTeX, in which case you should also submit electronically your source code). Resources on how to use LaTeX are available on the course's website. There will be a 5% bonus for electronically prepared answers (e.g., on MS Word, etc.) that are not typeset. If you want to submit a handwritten report, scan it and submit a PDF via Sakai. We will not accept hard-copies. If you choose to submit handwritten answers and we are not able to read them, you will not be awarded any points for the part of the solution that is unreadable.

Precision: Try to be precise. Have in mind that you are trying to convince a very skeptical reader (and computer scientists are the worst kind...) that your answers are correct.

Collusion, Plagiarism, etc.: Each team must prepare its solutions independently from other teams, i.e., without using common notes, code or worksheets with other students or trying to solve problems in collaboration with other teams. You must indicate any external sources you have used in the preparation of your solution. Do not plagiarize online sources and in general make sure you do not violate any of the academic standards of the course, the department or the university (the standards are available through the course's website: `http://www.pracsyslab.org/cs440`). Failure to follow these rules may result in failure in the course.

Markov Decision Process for Jerry Escaping Tom

In this assignment, you will use some of the algorithms for computing optimal policies in Markov Decision Processes (MDP's) to help Jerry the mouse escape from Tom the cat, while trying to find cheese to eat. The main components of the assignment are the following:

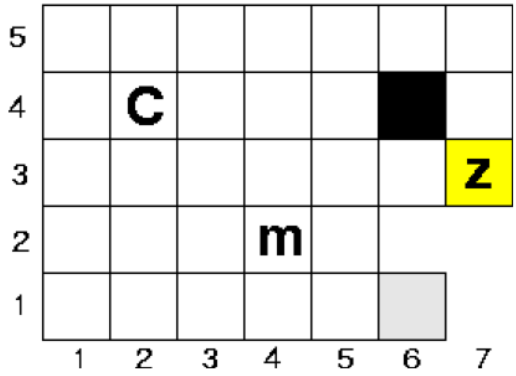
1. implement the policy iteration and value iteration algorithms for computing optimal policies, as well as a policy evaluation algorithm;
2. and run your code and explore its performance on a simulated cat-and-mouse world.

Implementing MDP algorithms

The first part of the assignment is to implement algorithms that were discussed in lecture, and which are also available in your textbook (sections 17.2 and 17.3): value iteration, policy iteration and policy evaluation. Your implementations should work for any (reasonable) MDP, not just the ones provided. See below for a description of where you should write your code.

Helping Jerry

The second part of the assignment is to use your code to compute optimal policies for a mouse who lives with a cat in the following grid world:



The mouse (m) can occupy any of the 31 blank squares. The cat (C) also can occupy any square, except for square (6,1) which is Jerry's hole (too small for the cat to squeeze in). There is cheese (z) that usually can be found in one of the following three squares: (2,3), (4,1) and (7,3). Occasionally, the cheese disappears altogether. Thus, this MDP has 31*30*4=3720 states.

Tom and Jerry can each move one square in any direction - vertically, horizontally or diagonally. They can also choose not to move at all. Thus, there are nine possible moves from each square. If an action is attempted that causes the creature to bump into a wall, then it simply stays where it is.

In this problem, we will always take the point of view of Jerry. When Jerry is on the cheese, it receives a reward of +1. When Tom is in the same cell with Jerry, Jerry receives a reward of -21 (ouch!). When Tom, Jerry and the cheese are in the same cell, the reward is -20. All other configurations have a reward of 0. Thus, Jerry is trying to eat cheese while simultaneously avoiding Tom.

We will consider three different versions of Tom:

1. The first cat, poor thing, is blind and oblivious, and simply wanders randomly around its environment choosing randomly among its nine available actions at every step.
2. The second version of Tom is hungry, alert and unrelenting. This cat always heads straight for the mouse following the shortest path possible. Thus, after the mouse makes its move, the

cat chooses the action that will move it as close as possible to the mouse's new position. (If there is a tie among the cat's best available options, the cat chooses randomly among these equally good best actions.) However, when the mouse is in its hole and out of sight, the cat reverts to aimless (i.e., random) wandering.

3. The third cat is also alert, but has a more sporting disposition, and therefore follows a combination of these two strategies: half the time, it wanders aimlessly, and half the time, it heads straight for the mouse.

Machine-readable (and compressed) descriptions of the three MDP's corresponding to these three cats is provided in the files "oblivious.mdp.gz", "unrelenting.mdp.gz" and "sporting.mdp.gz", under the data directory of the provided software package.

If there is cheese available, it will always be in exactly one of the three locations listed above. At every time step, the cheese remains where it is with 90% probability, and with 10% probability, it vanishes altogether. Once it has vanished, the cheese remains hidden at each step with probability 70%, and with 30% probability, it reappears randomly in one of the three designated locations.

States are encoded as six tuples, the first two numbers indicating the position of Jerry, the second two numbers the position of Tom, and the last two numbers the position of the cheese. Thus, 4:2:2:4:7:3 indicates, as depicted in the figure above, that Jerry the mouse (m) is in (4,2), that Tom the cat (c) is in (2,4), and the cheese (z) is in (7,3). When there is no cheese, the last two coordinates are empty; for instance, 4:2:2:4:: would encode the same state as above with the cheese missing.

Tom and Jerry alternate moves. Nevertheless, in encoding the MDP, we collapse both moves into a single state transition. In addition, the cheese, when it vanishes or reappears, does so simultaneously with the mouse's move. For instance, from the configuration above, if the mouse moves to (5,2) and the cat responds by moving to (3,3), while the cheese vanishes, this all would be encoded as a single transition from state 4:2:2:4:7:3 to 5:2:3:3::. Actions in the MDP refer to actions that the mouse can make; the cat's actions are effectively "hard-wired" into the dynamics of the MDP itself.

For each of the three versions of Tom, your job will be to compute the mouse's optimal policy, i.e., the action that should be taken at each state to maximize the mouse's expected discounted reward, where we fix the discount factor (gamma) to be 0.95. You can then watch the cat and the mouse go at it using the provided visualization.

Exploring Optimal Behavior

Once you have everything working, you should take some time to observe and understand the mouse's behavior in each of the three cases. Then answer the following:

1. Give some examples of how the mouse appears to be behaving intelligently. Try to point out any strategies you noticed that seemed especially surprising or non-obvious.

2. Considering what you understand about the algorithms that you just implemented, where is the mouse's apparent intelligence coming from? In your opinion, does the mouse actually possess "true" intelligence? Why or why not?

3. If you were directly controlling the mouse's actions at each time step, do you think it would be possible (in principle, perhaps with a lot of practice) for you to do a better job of getting cheese while not being eaten? Why or why not?

4. Give some specific examples of how this simulated world differs from a real-world situation involving real cats and mice. Suppose we wanted to "scale up" the MDP-based approach used in this assignment to more realistic situations that address some or all of these differences. What would be some of the difficulties that would need to be overcome to make this possible?

5. How do your implementations of policy iteration and value iteration compare in performance on MDP's of this kind? There are many ways of judging performance, and to fully answer this

question, you should consider more than one. For example (and these are only examples – you certainly are not required to answer all of these, and there are plenty of other possibilities not listed here), you might consider questions like: Which algorithm is faster? How many iterations does it take to reach convergence (which can be measured in more than one way)? How is performance affected if we stop iterating one of the algorithms before we reach convergence? What happens if we vary gamma?

The Provided Code

You have access to a class called "Mdp" that loads a description of an MDP from a file whose format is described shortly. The class has a constructor taking a file name as argument that reads in the MDP description from the file. Each state is represented by an integer in the range 0 (inclusive) to numStates (exclusive), where numStates is the total number of states. Actions are represented in a similar fashion. The stateName and actionName arrays convert these integers back to strings in the obvious fashion. The reward array gives the immediate reward of each state. For efficiency, state transition probabilities are represented in a sparse fashion to take advantage of the fact that most states have a relatively small number of possible successors. In particular, the array `nextState[s][a]` represents a list of possible next states, which may follow state s under action a. The understanding here is that the probability of transitioning to any other state is zero. There is a corresponding list of probabilities `transProb[s][a]`. Thus, `transProb[s][a][i]` is the probability of transitioning from s to `nextState[s][a][i]` under action a.

Each line of an MDP description file has one of three possible formats. The first possibility is that the line has the simple form: `<state>`
where `<state>` is any name (no white space). Such a line indicates that `<state>` is the start state of the MDP. The second possibility is that the line has the form:
`<state> <reward>`,
where `<state>` is any name, and `<reward>` is a number. Such a line indicates that the reward in `<state>` is `<reward>`.

The last possibility is that the line has the form:
`<from_state> <action> <to_state> <prob>`
where the first three tokens are names, and `<prob>` is a nonnegative number. Such a line indicates that the probability of transitioning from `<from_state>` to `<to_state>` under action `<action>` is `<prob>`. Multiple lines of this form with the same `<from_state>` and `<action>` can be combined. Thus:

$$u\ a\ v\ 0.3\ u\ 0.4\ w\ 0.1$$

is equivalent to the three lines:

$$u\ a\ v\ 0.3$$
$$u\ a\ u\ 0.4$$
$$u\ a\ w\ 0.1$$

Lines of these forms can appear in the file in any order. However, if a reward is given more than once for the same state, then the last reward value given is assigned to that state. On the other hand, if more than one probability is given for the same state-action-state triple, then the sum of these probabilities is assigned to this transition. If the probabilities of transitioning out of a given state under a given action do not add up to one, then these numbers are renormalized so that they do. However, if this sum is zero, an exception is thrown. States that are not given a reward in the description file are assumed to have a reward of zero; likewise, transitions that do not appear in the file are assumed to have zero probability. An exception is also thrown if no start state is declared (using the first form above); if more than one such line appears, then the last one is taken to define the start state.

For instance, the following encodes an MDP with five states, -2, -1, 0, +1 and +2. There are two actions, L(eft) and R(ight) which cause the MDP to move to the next state to the left or right 90% of the time, or to move in the opposite direction 10% of the time (with movement impeded

at the end points -2 and +2). The start state is 0. The reward is 1 in 0, -1 in -2, and -2 in +2. The start state is 0.

```
0
0 1
0 L −1 0.9 +1 0.1
0 R +1 0.9 −1 0.1
−1 L −2 0.9 0 0.1
−1 R 0 0.9 −2 0.1
−2 −1
−2 L −2 0.9 −1 0.1
−2 R −1 0.9 −2 0.1
+1 R +2 0.9 0 0.1
+1 L 0 0.9 +2 0.1
+2 −2
+2 R +2 0.9 +1 0.1
+2 L +1 0.9 +2 0.1
```

If you want to try out or test your program on MDP's of your own invention (which is highly recommended - please report that in your submission), you can do so by creating files of this form. Alternatively, you can create a subclass of the Mdp class, and write code that fills in the public fields with a description of your MDP.

Because the files being provided for the Tom and Jerry MDP's are quite large, they are provided in a compressed (gzipped) form. Note that the Mdp class will automatically read a file that has been gzipped, provided that it ends with a ".gz" suffix. In other words, there is no need to uncompress the gzipped files that are provided prior to running your code on them. (Of course, the code will also work on files that have not been compressed.) In addition, because these MDP's are somewhat large, you may need to increase the memory available to java (the option for doing this is system dependent, but it often has the form -Xmx256m which increases the memory to 256 megabytes, etc.).

The class "CatMouseAnimator" can be used to animate a given policy on one of the cat-and-mouse MDP's (and will surely crash if provided with any other kind of MDP). More precisely, the animator will animate a sequence of states provided to it by an object implementing the MdpSimulator interface; typically, these state sequences will be generated according to a fixed policy using the "FixedPolicySimulator" class.

The method "animateGuiOnly" invokes a graphical user interface (GUI) for this purpose showing the behavior of the cat and mouse in a graphical depiction of their grid-world, similar to the one above. Buttons are provided for starting or stopping the animation (at a speed that can be controlled), or for stepping through the animation one step at a time.

The method animateGuiAndPrint invokes the GUI as well, but also prints the state of the MDP at every step. The printed form of each state looks something like the one in Fig. 1.

As in the GUI, the letters *m*, *C* and *z* indicate the position of the mouse, cat and cheese. The mouse's hole is indicated with an underscore. In both the printed and GUI depictions, the mouse vanishes when the cat is on top of it; likewise, the cheese can be obscured by either the cat or the mouse (although the mouse changes from lower to upper case *M* when it is eating).

A final method animatePrintOnly prints the successive states of the MDP, but does not invoke the GUI.

The class "RunCatMouse" consists only of a main that loads an MDP from a file, finds and prints the optimal policy (using both value and policy iteration) and runs the animator (depending on command line arguments – see the documentation). Before starting the animator, four columns

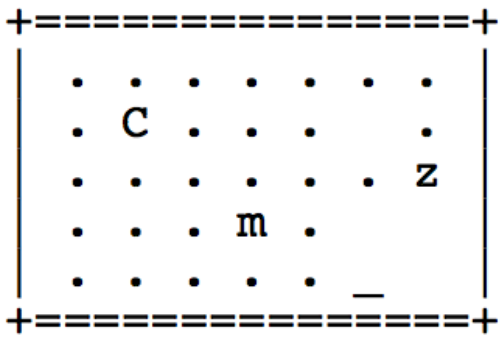


Figure 1: Example output

are printed following each state: the optimal action under the policy found using value iteration; the optimal action under the policy found using policy iteration; the optimal utility as estimated using value iteration; and the estimated utility of the optimal policy using policy evaluation. For instance, on the simple five-state MDP given above, the output should look something like this:

```
Optimal policies:
0    L    L    8.169018052871904    8.169018052871904
-1   R    R    7.5578670677920545   7.5578670677920545
+1   L    L    7.442544315827016    7.442544315827016
-2   R    R    6.035332975545492    6.035332975545492
+2   L    L    4.821409270650342    4.821409270650342
```

In this case, as expected, the optimal policy says to move to the center (right if in -1 or -2, and left if in +1 or +2), and to move left (away from the larger penalty state +2) if in the center.

(Note that your own results, even for a correct implementation, may differ slightly from the utilities that are given here, but should be accurate up to numerical precision.)

You might well decide to modify or replace the RunCatMouse class. However, your code should still work properly when running using the provided version.

To debug your code, you will surely want to run it on small MDP's. There are a number of "sanity checks" that can be done to see if your code is working. For instance, value iteration and policy iteration should output identical policies (up to possible ties between equally good actions). In addition, the utility of the optimal policy must satisfy the Bellman equations.

All code and a data files can be obtained from Sakai as a single zip file. Data is included in the "data" subdirectory.

As a general reminder, you should not modify any of the code that we are providing, other than template files, or unless specifically allowed in the instructions. You also should not change the "signature" of any of the methods or constructors that you are required to write, i.e., the parameters of the method (or constructor), their types, and the type of the value returned by the method.

The code that you need to write

Your job is to fill in the constructors in the three classes "ValueIteration", "PolicyIteration" and "PolicyEvaluation". The constructor for "ValueIteration" takes as input an MDP and a discount factor; the constructed class then includes a field policy containing the optimal policy (an array mapping states to actions), and another field utility (an array mapping states to real numbers). The other two classes are set up similarly.

You may wish to add other fields that are returned as part of the class, such as the number of iterations until convergence. You also may add other constructors or methods, provided that these are in addition to the ones given in the template files (which are the ones that will be automatically tested). In particular, your code must work properly when run with the main provided in

“RunCatMouse.java”.

Your implementation of policy evaluation should compute the utility of a given policy for every possible starting state s. Unlike the description of policy evaluation given in your textbook, the version that you are asked to write should not take as input a starting estimate of the utility. (As noted above, you may add a constructor or method that takes such a starting estimate, but this must be in addition to the constructor that we are providing in the template file.) Your implementation should compute an estimate of the utility of the given policy that is at least as accurate as specified below.

As discussed in class and in your textbook, there are at least two techniques for evaluating a policy. In the first approach, we use methods from linear algebra to solve a set of linear equations that are a simplified form of the Bellman equations. In the second approach, we use an iterative procedure to solve these equations that is essentially a simplified form of value iteration; thus, successive estimates of the utility are repeatedly plugged into the (simplified) Bellman equations to obtain new estimates, a process that terminates when the change in the estimates is small enough to guarantee that the utility estimates will meet the specified accuracy requirements. Because the linear-algebraic approach is fairly expensive, you should use the second approach on this assignment.

Your policy-iteration code should make use of your implementation of policy evaluation, on each iteration obtaining an exact (up to the specified accuracy) estimate of the utility of the current policy.

To clarify how this relates to the presentation of policy iteration in your textbook: The book describes both "standard" policy iteration, and a modification called "modified" policy iteration. The distinction is in how policy evaluation is implemented – in the standard version, policies are evaluated exactly, while in the modified version, they are only approximately evaluated using just a few steps of simplified value iteration. For this assignment, your implementation should be a version of standard policy iteration in the sense that you should evaluate policies (nearly) exactly. However, to do so, you should be using the iterative approach described above; this amounts to a procedure similar to modified policy iteration except that, in evaluating a policy, instead of running for a fixed number of simplified value iteration steps, your code should run until the change in successive approximations of the policy’s utility is sufficiently small.

The utilities computed by your implementation of value iteration and policy evaluation should, at every state, be within 10-8 ($1e^{-8}$) of the true utility for that state. In other words, if utility is the array of (estimated) utility values returned by `ValueIteration`, then for every state s, `utility`[s] must be within 10^{-8} of the true optimal utility at state s. Likewise, for `PolicyEvaluation`, `utility`[s] must be within 10^{-8} of the utility of state s under the given policy.

The final code that you turn in should not write anything to standard output or standard error. Your code must work properly when run with the provided classes on the provided MDP’s.

Your own problem

For this component of the assignment, you are asked to pick a non-trivial problem, formulate it as a MDP and solve it. What we are looking for is the creativity in the design of your MDP and effectiveness of your implementation to solve the MDP. The amount of HP points you will receive will heavily depend on how interesting your problem is and whether you get the solution correctly and efficiently.

1. Describe a non-trivial problem and formulate as a MDP. (Specify the states, actions in each state, a transition model, a reward function and the discount factor, etc..)
2. Write the code needed for both generating your MDP and solving it. For this problem, you may use and modify any given code from us for the Tom-and-Jerry problem as well as your implementation for value iteration, policy iteration and the modified policy evaluation algorithm. You need to name one class RunHPMdp which has a main function for running your implementation on your MDP included in a file named HPMdp.txt. Therefore, on a Unix system, the command "java RunHPMdp HPMdp.txt" will run the program on the MDP in the file HPMdp.txt.

3. Report on whether your implementation finds correct solutions to the MDP provided in HP-MDdp.txt? How do you know whether the solution is correct or not? How much time did it take to solve your MDP? If you were given more time, what would you do to improve your implementation or your problem design?

4. For the real world problem you picked, is formulating it as a MDP the best way to solve it? Why or why not?

The TAs will judge your solution based on the code, the report and the output of your program. While you may not need to animate a given policy on your MDP on a GUI, a method that prints the state of your MDP at every step should be helpful.

What to turn in

On Sakai, you should turn in the following:

- The classes ValueIteration.java, PolicyIteration.java and PolicyEvaluation.java with the constructors filled in.
- Any other java files that you wrote and used, or that are needed by your code.
- A readme.txt file explaining briefly how your code is organized, what data structures you are using, or anything else that will help the TA understand how your code works.
- In addition, you should turn in your program report exploring the Tom-and-Jerry environments as described above.

If you solve your own problem, you should turn in the following:

- RunHPMdp.java and all other java files needed (includes the code you wrote and the code provided by us) to solve the problem.
- HPMdp.txt with the MDP of your problem
- A readme.txt file that briefly explains how your codes are organized and how the MDP (i.e. HPMdp.txt) is generated.
- A reportHP.pdf with answers to the questions 1, 3, and 4 from the HP section. Your report for this HP problem should not exceed two pages.

What you will be graded on

Your code will be tested on MDP's other than those provided with this assignment. Your grade will depend largely on getting the right answers. In addition, your code should be efficient enough to run reasonably fast (under a minute or two on a fast machine on an MDP like those provided), and should not terminate prematurely with an exception (unless given bad data, of course); code that does otherwise risks getting no credit for the automatic testing portion of the grade. As usual, you should follow good programming practices, including documenting your code.

Your program report should be clear, concise, thoughtful and perceptive.

There will be 60 points (50 for graduates) based on automatic testing of your code, efficient implementation of the correct algorithms, proper documentation and good programming practice, and 40 for the report. Generating and submitting your own MDP problem is worth 20 points (depending on creativity, correctness, completeness, etc.).