

Massachusetts Institute of Technology

16.410-13 Principles of Autonomy and Decision Making

Problem Set #1

Due: Session 4

Objective

The primary objective of this problem set is to begin to exercise your skill at designing and implementing Java programs. This includes developing comfort at programming in Java and using Junit to test your code. You will also become familiar with queue implementations, which you will use in Problem Set #2. In addition, the second objective of this problem set is to test your understanding of state space problem formulation, and the process of uninformed search.

Background

In the following, JINS denotes “Java in a Nutshell,” by David Flanagan, while AIMA denotes “Artificial Intelligence: A Modern Approach,” by Russell and Norvig.

For objective one, start by reading the intro, chapter one, of JINS. Next, run the chapter’s factorial example using the Java Developers Kit (JDK) and the Eclipse Integrated Development Environment (IDE), as described in the class handout, “Java Jump Start.” This will give you a sense of the Java debugging and development cycle.

Next, to develop your understanding of the Java language, work through the Sun Java tutorial, located at <http://java.sun.com/docs/books/tutorial/information/download.html>. In addition, you may want to read Chapter 3 of JINS to further develop your understanding of object oriented programming in JAVA, and Chapter 2 of JINS to develop your understanding of expressions in the language.

You will save significant time implementing Problems 1-3 by using the Collections package, which is part of the java.util package. This is described in JINS pages 224-238. In addition, you will likely find the Sun online API documentation useful for this (<http://java.sun.com/j2se/1.4.2/docs/api/index.html>).

You can choose to develop and debug your code using command-line commands, through JDK, or using the window-based Eclipse IDE. Both can be found on the lab machines. Many of you will want to load these systems on your own machines. You can download Java and JDK (j2sdk-1_4_2_09-windows-i586-p.exe) from

<http://java.sun.com/j2se/1.4.2/download.html>. You can download Eclipse (eclipse-SDK-3.1-win32.zip) from <http://eclipse.org/downloads/index.php>.

JDK is described in Chapter 8 of JINS. Of particular note is the command “javac,” which compiles a java source file (.java) to byte code file (.class), the command “java,” which interprets (runs) a byte code file, and the command “jdb,” which is used to debug a java program. Eclipse provides an online tutorial and online documentation, by starting up Eclipse, and clicking Help in the toolbar.

For objective two, read Chapters 2 and 3 of AIMA, while paying careful attention to the vacuum World problem. Uninformed search is covered in Chapter 3 of AIMA, and the notes for lecture 3.

A Special Note About Junit and Automatic Grading

Please take care to read the instructions carefully. If you are asked to implement a class called “Foo,” it is important that your class be called “Foo”. If you do not do this, then our automated test code will fail to find your class and you will get no credit for your effort. Similarly, method signatures must match those specified in the problem set.

Automated testing is performed using Junit. You can download Junit (junit3.8.1.zip) from <http://www.junit.org/index.htm>.

There is lots of other good stuff on www.junit.org check out their “Documentation” tab. There are some simple examples of how to use Junit. Junit is fun to play with.

Problem 1

In this problem you will begin exercising your skill at simple Java programming, debugging (via JDK or Eclipse) and testing (using junit). You should begin by learning about the `java.util.LinkedList` class and the interface `java.util.List` (the API documentation mentioned above or JINS will be useful for this).

Implement a class called `FunWithLinkedLists` that does the following:

1. Implements a reverse method that takes a `LinkedList` and returns a new `LinkedList` that has the same elements as the old `LinkedList`, but in the reverse order. Your method signature must be
`{public LinkedList reverse (LinkedList ll);`
2. Implement an append method that takes two `LinkedList` arguments and returns a new `LinkedList` which has all the elements of the first `LinkedList` followed by all of the elements of the second `LinkedList`. The method signature must be
`{public LinkedList append (LinkedList a, LinkedList b);`

Build a Junit test suite for the new methods, `reverse` and `append`, that you developed in Steps 1 and 2. Demonstrate, by showing the output from Junit, that your new methods work correctly.

Problem 2

Create a file called Queue.java, comprised of the following text:

```
import java.util.List;

public interface Queue {
    void add (Object anItem);
    void add (List items);
    Object remove();
    Object get();
}
```

Next, implement two classes as follows:

```
public class FIFOQueue1 implements Queue {
    protected LinkedList linkedList;

    public FIFOQueue1() {
        linkedList = new LinkedList();
    }

    // ***Add any helper methods here

    public void add (Object anItem) {
        // ***Add implementation code here
    }

    public void add (List items) {
        // ***Add implementation code here
    }

    public Object remove () {
        // ***Add implementation code here
    }

    public Object get () {
        // ***Add implementation code here
    }
}

public class LIFOQueue1 implements Queue {
    protected LinkedList linkedList;

    public LIFOQueue1() {
        linkedList = new LinkedList();
    }

    // ***Add any helper methods here

    public void add (Object anItem) {
```

```

    // ***Add implementation code here
}

public void add (List items) {
    // ***Add implementation code here
}

public Object remove () {
    // ***Add implementation code here
}

public Object get () {
    // ***Add implementation code here
}
}

```

You should put these classes in file with the same name as the class with a “.java” extension. (FIFOQueue1.java and LIFOQueue1.java). When a class is compiled it creates one “.class” file for each class it encounters.

Hint: You will likely find it convenient to use your “reverse” and “append” methods in these classes, from Problem 1.

A LIFO queue is a queue in which the last element to be added to the queue is the first element to come out of the Queue. Contrariwise a FIFO is a conventional queue in which the first one in is the first one out.

Write Junit tests for LIFOQueue1 and FIFOQueue1, and use them to demonstrate that the two classes function correctly.

Problem 3

The definitions of FIFOQueue1 and LIFOQueue1 are hard to read, and may involve some duplication of code. The difficulty with reading comes from the list manipulation that must occur in the implementation and the duplication of code comes from the similarity of the operations that must occur for the FIFO and the LIFO queues. Fix these problems by implementing an abstract queue called AbstractQueue and by implementing FIFOQueue2 and LIFOQueue2 as sketched below.

```

public class AbstractQueue implements Queue {
    protected LinkedList linkedList;

    public AbstractQueue() {
        linkedList = new LinkedList();
    }

    // ***Add any helper methods here

    public void add (Object anItem) {

```

```
        throw new RuntimeException("must be implemented by
subclasses");
    }

    public void add (List items) {
        throw new RuntimeException("must be implemented by
subclasses");
    }

    public Object remove () {
        throw new RuntimeException("must be implemented by
subclasses");
    }

    public Object get () {
        throw new RuntimeException("must be implemented by
subclasses");
    }
}
```

```
public class FIFOQueue2 extends AbstractQueue {

    public void add (Object anItem) {
        // ***Add implementation code here
    }

    public void add (List items) {
        // ***Add implementation code here
    }

    public Object remove () {
        // ***Add implementation code here
    }

    public Object get () {
        // ***Add implementation code here
    }
}
```

```
public class LIFOQueue2 extends AbstractQueue {

    public void add (Object anItem) {
        // ***Add implementation code here
    }

    public void add (List items) {
        // ***Add implementation code here
    }

    public Object remove () {
        // ***Add implementation code here
    }
}
```

```

    }

    public Object get () {
        // ***Add implementation code here
    }
}

```

As with LIFOQueue1 and FIFOQueue1, write Junit tests for LIFOQueue2 and FIFOQueue2 and demonstrate that they function correctly.

Hint: Since LIFOQueue1 has the same semantics as LIFOQueue2 and FIFOQueue1 has the same semantics as FIFOQueue2, the test suite for LIFOQueue1 and FIFOQueue1 can be used for LIFOQueue2 and FIFOQueue2.

Problem 4 Compact Encoding of a State Space using Variables

In this problem we walk through a more precise description of states and operators in a state space problem, by using variables. Consider the Vacuum World problem introduced in Chapters 2 and 3 of AIMA. We assume that the suction unit is fully functional. In the following we will use state variable assignments to compactly encode both states, and operators that transition between states.

The *states* of a problem can be viewed as a finite set of unique symbols. In this problem, to represent these states compactly we encode them using a set of *state variables*, each of which has a corresponding *domain*. A domain is a set of possible values that a variable can take on. In this formulation, a state is a particular choice of values for each of the state variables.

For example, let A and B be two variables, with domains {yes, no} and {red, blue}, respectively. Note that we use curly brackets, “{ }”, to delimit a set of elements. An example state under this encoding is {A = yes, B = red}, comprised of an assignment to each of A and B. The complete set of states under this encoding is:

$$\{ \{A = \text{yes}, B = \text{red}\} \{A = \text{no}, B = \text{red}\} \{A = \text{yes}, B = \text{blue}\} \{A = \text{no}, B = \text{blue}\} \},$$

which is comprised of four states.

Note that the order in which variables are assigned does not matter; any two orderings that are comprised of identical assignments denote the same state. For simplicity, we will often assume a fixed ordering of the state variables, and will then exploit this to describe a state simply by listing the values of the state variables in order. For example, we will assume the variable ordering <A, B>. Note that angle brackets, “<>”, are used to delimit an ordered list of elements, in this case “A” followed by “B”. Given this ordering, the state {A = yes, B = red} is denoted <yes, red>.

In this problem we will consider two different encodings for the Vacuum World problem. The first encoding, State Encoding 1, is comprised of the following four variables and their corresponding domains:

Left: {dirty, empty}
Right: {dirty, empty}
Vacuum: {left, right}
Suction: {on, off}

To denote a state we assume the variable ordering $\langle \text{Left}, \text{Right}, \text{Vacuum}, \text{Suction} \rangle$. Hence the state denoted by $\langle \text{dirty}, \text{empty}, \text{left}, \text{off} \rangle$ is equivalent to:

{Left = dirty, Right = empty, Vacuum = left, Suction = off}

The second variable encoding, State Encoding 2, is:

Dirt: {none, left, right, both}
Vacuum: {left, right}
Suction: {on, off}

Having encoded the state space, we now turn to encoding the operators. An *operator* is a function that performs a transition between two states, a source and a target. Operators are defined by enumerating the state transitions. Each transition is denoted by a rule of the form “*source* -> *target*.” For example, consider the following transition for the move left operator:

$\langle \text{empty}, \text{empty}, \text{right}, \text{off} \rangle \rightarrow \langle \text{empty}, \text{empty}, \text{left}, \text{off} \rangle$

This transition says that, starting in the source state in which Left and Right are empty, the Vacuum is to the right, and Suction is off, the result of applying the move left operator to this source is the target state in which the Vacuum variable has become left, and the assignments to the other three variables are unchanged.

Often the assignment to one or more variables in the source state is irrelevant to the target state. In this case, we use the value “?” in a source position to denote that the corresponding state variable assignment is irrelevant. If “?” also appears in the target position for this “irrelevant” variable, then the operator does not change the variable value, and the source value is copied to the target. For example, the following expression for move-left:

$\langle ?, ?, \text{right}, \text{off} \rangle \rightarrow \langle ?, ?, \text{left}, \text{off} \rangle$

says that the vacuum will move from right to left independent of whether or not there is dirt on the Left or Right, and the Left/Right dirt states will remain unchanged, given that the Suction is off.

Note that an operator may not be defined for all source states. In this case we assume that an operator can only be applied if the system is currently in a source state that matches the left hand side of one of the operator’s rules.

For the vacuum world, using State Encoding 1, we encode the four operators as follows:

```
move-left:
    <?, ?, right, off> -> <    ?, ?, left, off>
    <?, ?, right, on> -> <empty, ?, left, on>

move-right:
    <?, ?, left, off> -> <?,    ?, right, off>
    <?, ?, left, on> -> <?, empty, right, on>

turn-on:
    <?, ?, left, off> -> <empty,    ?, left, on>
    <?, ?, right, off> -> <    ?, empty, right, on>

turn-off:
    <?, ?, ?, on> -> <?, ?, ?, off>
```

- Generate similar operator descriptions for the four Vacuum world operators using State Encoding 2. Is your description more compact than the operator description that we provided for State Encoding 1? Please explain why.
- Note that Encoding 2 has less state variables than Encoding 1, but more values in the largest domain. In what situations is Encoding 1 better? In what situations is Encoding 2 better?
- How many bits does it take to encode a state under each of the given representations (if you don't know what a bit is, you can think of it as a Boolean variable that can take on one of two values, "true" or "false")? Is this the most compact encoding that you can provide?
- Assume that we are using the state and operator description for Encoding 1, above, and are performing a breadth first search. We start with the state `<dirty, dirty, right, off>` and our goal is a state of the form `<empty, empty, ?, off>`. List the states that are added to the search queue at each step, stopping when the goal is found. Draw the search tree up to the point where the goal has been found. Use the format for search queues and search trees given in the notes for lecture 3.

Note: Developing this queue should not take long and is not that difficult, but you will want to ensure that you understand it completely, and that you don't make simple mistakes (the devil is in the details here). Remember that in each step of the search, we pull a state off of the queue, expand it out to new states, and put these new states on the queue. One tricky point is that we do not stop until we pull a goal state off the queue – not when we first put it on the queue. Another is that we do not allow any loops in a path – so no path down the tree can reach the same state twice.

Problem 5 Time

Please list the amount of time that you devoted to completing this problem set.