THE UNIVERSITY OF BIRMINGHAM

Degree of B.Sc.
Artificial Intelligence and Computer Science. Second and Final Examinations.
Bioinformatics. Second Examination.
Computer Science. Second and Final Examinations.

Joint Degree of B.Sc. with Honours
Mathematics and Artificial Intelligence. Second Examination.
Psychology and Artificial Intelligence. Second Examination.

Degree of B.Eng./M.Eng.
Computer Science/Software Engineering. Third and Final Examinations.

Joint Degree of B.A. with Honours
Arts and Artificial Intelligence. Second Examination.
Arts and Computer Science. Second Examination.
Natural Sciences. Second Examination.
Occasional: American Studies.

06 08163

Software Workshop Java

May 2003  2 hours

[Answer ALL questions]
Please answer all questions on the pre-formatted answer sheet provided. If you need more space for any particular question, you can continue in a standard answer book. Please use a treasury tag to attach this sheet to the inside of your answer book.

1. A graph showing airline costs between some European cities is shown below. Dijkstra’s algorithm may be used to find a route of smallest cost between two cities. The algorithm maintains an instance of the class RouteTableau. Some of the code of RouteTableau extracted from the solution of the coursework exercise is shown at the end of this question (slightly modified).

![Airline Cost Graph](image)

RouteTableau is used as follows. An instance is created when its constructor is called with the origin. Then, the extend() method is called repeatedly, until it returns false. Finally, the getItinerary() method is called to extract the itinerary.

The extend() method extends the tableau, returning a boolean indicating whether extension was possible. This means:
Select the line (CITY, FROM, COST, CHEAPEST) with smallest COST which is not marked as cheapest (lines 24–28);  

For each leg from CITY,  

if the leg’s destination is already in the tableau (line 48), then  

if this leg represents a way of getting there at a smaller cost,  

adjust COST and FROM accordingly (lines 51–53),  
else  

do nothing (the tableau already has a cheaper way of getting to destination)  
else  

add this way of getting to destination to the tableau (lines 57–60).  

Return a boolean saying whether extension of tableau possible.  
(The caller method’s loop will terminate when we return false.)  

For example, the tableau for the route from Edinburgh to Marseille is computed as follows:  

Constructor called; the tableau becomes  

to:          from:          totalCost  cheapest?  
Edinburgh    null           0.0          false  

The city selected is Edinburgh, and extend() called; the tableau becomes  

to:          from:          totalCost  cheapest?  
Paris        Edinburgh     120.0        false  
Birmingham   Edinburgh     80.0         false  
Edinburgh    null           0.0          true  

The city selected is Birmingham and extend() called again; the tableau becomes  

to:          from:          totalCost  cheapest?  
Paris        Edinburgh     120.0        false  
Hannover     Birmingham    260.0        false  
Birmingham   Edinburgh     80.0         true  
Edinburgh    null           0.0          true  
London       Birmingham    160.0        false  

Note that extending the tableau at the Birmingham node does not cause Edinburgh to be entered (it is already present) and does not cause the Edinburgh entry to be modified (it is already optimal).  

The extend() method is called 4 more times, expanding the tableau at Paris, Hannover, London and Lyon, at which time the method extend() returns false and the tableau is:
to:       from:       totalCost  cheapest?
Paris  Edinburgh  120.0  true
Lyon  Paris  220.0  true
Hannover  Paris  150.0  true
Birmingham  Edinburgh  80.0  true
Edinburgh  null  0.0  true
London  Birmingham  160.0  true
Marseille  London  280.0  false

(a) Use Dijkstra's algorithm to compute the smallest cost to travel from Marseille to Hannover. Give the tableau at each stage of the computation.  

(b) After the extend() method has been repeatedly called until it returned false, the getItinerary() method is used to extract the itinerary to dest and return it as an ArrayList of Strings. Give code for the getItinerary() method.  

(c) We now wish to modify the algorithm so that it computes the smallest cost to travel from a given city to any one of a set of cities. For example, if the origin is Birmingham and the destination set is {Marseille, Paris, Hannover} then the route Birmingham-London-Paris will be computed, with a cost of 130.

Briefly explain how to modify the extend() method so it takes an array of destinations as argument (state which lines of the extend method should be changed).  

(d) A travelling saleswoman starting from London wishes to visit Edinburgh, Hannover, Lyon and Marseille and then return to London. She does not mind in what order she visits the target cities, but wants to minimise the cost of her itinerary.

In an attempt to do this, she first uses the modified Dijkstra's algorithm to find the minimal route from London to one of {Edinburgh, Hannover, Lyon, Marseille}, say x. Next, she eliminates x from her target set and finds the minimal route from x to the new target set. She continues in this way until she has a route that visits all the cities. State what route she will find.  

(e) Does our travelling saleswoman always succeed in minimising the cost of her itinerary? If your answer is yes: explain why. If your answer is no: give an example situation where her method results in a non-minimal cost.  

```java
1 class RouteTableau {
2     HashMap lines;
3 }
4
class Detail {
5     String from;
6     float cost;
7     boolean cheapest;
8     public Detail(String f, float d, boolean c) {
9         from=f; cost=d; cheapest=c;
10     }
```
public boolean extend(String dest, RouteDatabase db) {
    // if tableau already has cheapest rt to destination, do nothing
    if (!lines.containsKey(dest))
        if (((Detail)lines.get(dest)).cheapest) return false;

    // select line in tableau with smallest cost not marked as cheapest
    String nextCity = this.findClosest();

    // if not successful, return false
    if (nextCity == null) return false;

    float cost = ((Detail)lines.get(nextCity)).cost;

    // mark selected line as known to be cheapest to that city
    ((Detail)lines.get(nextCity)).cheapest = true;

    // if that’s the destination, don’t bother to extend
    if (nextCity.equals(dest)) return false;

    // find legs starting at nextCity
    List newLegs = db.getLegs(nextCity);
    Iterator i = newLegs.iterator();

    // if none, return failure
    if (!i.hasNext()) return false;

    // for each leg, if destination already in tableau with at least
    // as good a cost, adjust cost and from; else add leg to the tableau
    while (i.hasNext()) {
        Leg leg = (Leg) i.next();
        if (!lines.containsKey(leg.getDestination())) {
            Detail d = (Detail) lines.get(leg.getDestination());

            // if leg is a better route to destination, replace tableau entry
            if (leg.getCost() + cost < d.cost) {
                d.cost = leg.getCost() + cost;
                d.from = leg.getOrigin();
            }
        } else
            lines.put(leg.getDestination(),
                new Detail(leg.getOrigin(),
                    leg.getCost() + cost,
                    false));

        return true;
    }

    // add a new entry
    }

    public ArrayList getItinerary(String dest) {
        return new ArrayList();
    }
2. Consider the code fragment from a solution for the exercise on predictive text, appearing at the end of this question. The class `WordSig` is an auxiliary class that holds a pair of strings, corresponding to a word and its numeric signature, on a device like a mobile phone. The class `PredText` contains

- an `ArrayList` object, `dictionary`, which contains `WordSig` objects and is sorted in the ascending order imposed by `WordSig`'s `compareTo` method,
- `digits`, a string that represents the sequence of digits that the user has typed (we will ignore sequences that contain spaces),
- `pos`, an integer representing the position in the `dictionary` of the current match for the entered `digits`.

The code for two methods of `PredText` is shown, `addDigit`, which is called by the user interface whenever the user presses a digit and `nextMatchingWord`, a method which is called when the user presses * and which looks up the dictionary for an alternative match of the digits already entered. Both methods return a string that represents the (possibly partial) word that matches `digits`.

Suppose that the program is already running and that an instance `pt` of `PredText` has been created. Its `dictionary` contains the following:

<table>
<thead>
<tr>
<th>i</th>
<th>sig</th>
<th>word</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>222</td>
<td>cab</td>
</tr>
<tr>
<td>1</td>
<td>24</td>
<td>ah</td>
</tr>
<tr>
<td>2</td>
<td>245</td>
<td>ail</td>
</tr>
<tr>
<td>3</td>
<td>24563</td>
<td>chloe</td>
</tr>
<tr>
<td>4</td>
<td>2456368</td>
<td>ailment</td>
</tr>
<tr>
<td>5</td>
<td>4475</td>
<td>girl</td>
</tr>
<tr>
<td>6</td>
<td>92837</td>
<td>water</td>
</tr>
</tbody>
</table>

Also, `digits` is set to the empty string, "", and `pos` is set to zero.

(a) The user presses a sequence of digits and the method `addDigit` is called repeatedly, with each digit as an argument. Execute the calls by hand and enter the values of `digits`, `pos` and the return value of the method in the table provided.

1. `addDigit('2')`
2. `addDigit('4')`
3. `addDigit('5')`
4. `addDigit('6')`
5. `addDigit('3')`
6. `addDigit('6')`
7. addDigit('8') [10%]

(b) Now, the user presses a sequence of digits and possibly the *-button which invokes the `nextMatchingWord` method. Execute the method calls shown in the leftmost column. Enter the values of `digits`, `pos` and the return value of the method in the table provided.

1. addDigit('2')
2. addDigit('4')
3. addDigit('5')
4. `nextMatchingWord()`
5. addDigit('6')
6. `nextMatchingWord()`
7. `nextMatchingWord()`
8. `nextMatchingWord()`
9. `nextMatchingWord()` [10%]

(c) As observed in the previous question, there is a bug in `nextMatchingWord()`. If a partial match has been found for the string of digits the user has entered, then pressing the *-button and thus calling `nextMatchingWord()` may return a word that is neither a partial nor a total match of the entered digits. State briefly what changes need to be made to the method `nextMatchingWord()` so that whenever `pos` points to a partial match for the digits entered and then `nextMatchingWord()` is called, it does not change `pos`. Indicate the lines of the method that should be changed. [10%]
class WordSig implements Comparable {
    String word, sig;
    public WordSig(String w, String s) { word=w; sig=s; }
    public int compareTo(Object ws){ return sig.compareTo(((WordSig)ws).sig); } 
    ...
    }
    
    public class FredText {
    private String digits;
    private ArrayList dictionary;
    private int pos=0;
    ...
    // called when the user presses digit c
    String addDigit(char c) {
        digits += c;
        pos = Collections.binarySearch(dictionary, new WordSig("",digits));
        // negative pos indicates signature not found; in this case,
        // -1-pos is the position at which the signature would be inserted
        if (pos < 0) pos = -1-pos;
        else {
            // there might be several matching words, backup to first one
            while(((WordSig)dictionary.get(pos)).sig.equals(digits) & & pos>0)
                pos--;
            if (pos < dictionary.size()-1) pos++;
        }
        WordSig found = (WordSig)dictionary.get(pos);
        String word;
        if (found.sig.startsWith(digits))
            word = found.word.substring(0,digits.length());
        else word="";
        
        return word;
    }
    // find and return next matching word
    String nextMatchingWord() {
        // normally, just increment pos
        if(pos < dictionary.size()-1) pos++;
        // if overshot, cycle round
        if(((WordSig)dictionary.get(pos)).sig.equals(digits)) {
            // we overshot, so backup
            pos--;
            while(((WordSig)dictionary.get(pos)).sig.equals(digits) & & pos>0)
                pos--;
            if(pos < dictionary.size()-1) pos++;
        }
        return (((WordSig)dictionary.get(pos)).word).substring(0, digits.length);
    }
"
3. A company wishes to issue invoices to its clients. The invoices will look like this:

<table>
<thead>
<tr>
<th>Description</th>
<th>Unit Price</th>
<th>Quantity</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Toaster</td>
<td>29.95</td>
<td>3</td>
<td>89.85</td>
</tr>
<tr>
<td>Hair Dryer</td>
<td>24.95</td>
<td>1</td>
<td>24.95</td>
</tr>
<tr>
<td>Car vacuum</td>
<td>19.99</td>
<td>2</td>
<td>39.98</td>
</tr>
<tr>
<td><strong>Total amount due:</strong></td>
<td></td>
<td></td>
<td><strong>154.78</strong></td>
</tr>
</tbody>
</table>

Three program skeletons for producing these invoices are shown at the end of this question. In Program 1, an invoice is created by the Invoice constructor, which takes the client to be invoiced as parameter. For each item to appear on the invoice, an instance of the Item class is created and the add() method of Invoice is called. The total price for an item is returned by the method getTotalPrice() in Item. This is defined to be the unitPrice multiplied by the quantity, together with salesTax if applicable. Sales tax is applicable to some clients but not others, as indicated by the boolean value salesTaxApplicable in Client. The getTotalPrice() method takes the client as a parameter so it can determine whether the totalPrice should have the sales tax added or not. The method getAmountDue() in Invoice calls getTotalPrice() for each item, and returns the sum of the values returned.

Program 2 is similar to Program 1, but differs in these respects.

- There is no add() method in Invoice. Instead, the Invoice constructor takes the items to appear on the invoice. The paper that the invoices are printed on is only big enough for five items, so there are never more than five items on the invoice. If we want an invoice with (say) three items, we simply set it4 and it5 to null.

- The getTotalPrice() method takes a boolean parameter indicating whether sales tax should be included, instead of taking the client as parameter.

(a) i. In respect of the way items are handled, which is better, Program 1 or Program 2? [4%]
   ii. Precisely state two arguments in support of your answer. [4%]

(b) i. In respect of the way sales tax is handled, which of Program 1 and Program 2 shows greater coupling between the classes? [4%]
   ii. Regarding the way sales tax is handled, what sort of programmer error due to greater coupling between classes could occur? [4%]
Program 3 is similar to Program 1, but with this difference:

- The program has been simplified by removing the class `Item`. Instead, the `Invoice` object has three `ArrayLists` of descriptions, quantities and unit prices. The method `getAmountDue()` in `Invoice` computes the total for each item by multiplying the quantity by the unit price and adding sales tax if applicable, and summing up the totals.

(c) i. Which version of `Invoice` shows greater cohesion, the one in Program 1 or Program 3? [4%]

ii. In this example, what are some disadvantages of low cohesion in the `Invoice` class? [4%]
/ PROGRAM 1

public class InvoiceProgram
{
    public void main()
    {
        Client cli = ...;
        Invoice inv1 = new Invoice(cli);
        inv1.add(new Item("Toaster", 29.95, 3));
        inv1.add(new Item("Hair dryer", 24.95, 1));
        ...
    }
}

class Invoice
{
    private Client client;
    private ArrayList items;

    public Invoice(Client c) { ... }
    public void add(Item item) { ... }
    public String format() { ... }
    public double getAmountDue() { ... }
}

class Item
{
    private String description;
    private int unitPrice;
    private int quantity;

    public Item(String descr, double pr, int quant) { ... }
    public int getQuantity() { return quantity; }
    public String getDescription() { return description; }
    public double getTotalPrice(Client c) { ... }
}

class Client
{
    private String name;
    private Address address;
    private boolean salesTaxApplicable;
    private double balance;
    private double creditLimit;

    public Client(String n, Address a, boolean tax) { ... }
    public void receivePayment(double amount) { ... }
    public void setCreditLimit(double amount) { ... }
    ...
}
// PROGRAM 2

public class InvoiceProgram {
    public void main() {
        Client c1 = ...;
        Invoice inv1 =
            new Invoice(c1,
                new Item("Toaster", 29.95, 3),
                new Item("Hair dryer", 24.95, 1),
                null, null, null);

        ... 
    }
}

class Invoice {
    private Client client;
    private ArrayList items;

    public Invoice(Client c, Item it1, Item it2, Item it3,
                    Item it4, Item it5) { ... }

    public String format() { ... }

    public double getAmountDue() { ... }
}

class Item {
    private String description;
    private int unitPrice;
    private int quantity;

    public Item(String descr, double pr, int quant) { ... }

    public getQuantity() { return quantity; }

    public getDescription() { return description; }

    public getTotalPrice(boolean incSalesTax) { ... }
}

class Client {
    private String name;
    private Address address;
    private boolean salesTaxApplicable;

    public Client(String n, Address a, boolean tax) { ... }

    public void receivePayment(double amount) { ... }

    public void setCreditLimit(double amount) { ... }

    ...
}
// PROGRAM 3

class Invoice
{
    private Client client;
    private ArrayList descriptions;
    private ArrayList quantities;
    private ArrayList unitPrices;

    public Invoice(Client c) { ... }
    public void add(String descr, double price, int quant) { ... }
    public String format() { ... }
    public double getAmountDue() { ... }
}

class Client
{
    private String name;
    private Address address;
    private boolean salesTaxApplicable;
    private double balance;
    private double creditLimit;

    public Client(String n, Address a, boolean tax) { ... }
    public void receivePayment(double amount) { ... }
    public void setCreditLimit(double amount) { ... }
    ... 
}