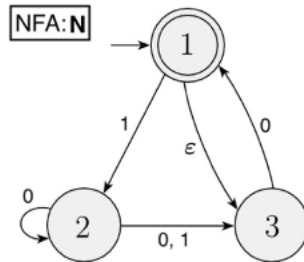


Assignment Summary:

$N = (Q, \Sigma, \delta, q_0, F) = (\{1, 2, 3\}, \{0, 1\}, \delta, \{1\}, \{1\})$, with δ :

	ϵ	0	1
1	3		2
2		{2, 3}	3
3		1	

We begin our assignment considering the NFA N . This solution implements a program to simulate the behavior of a nondeterministic finite automaton (NFA) on a given input string, building upon the first task, which involved reading strings for a deterministic finite automaton. Our solution adapts the approach by creating a tree of scenarios for the NFA using a queue and performing a breadth-first search after iterating through the input string. This allows us to check if any of the leaf nodes in the tree of scenarios are accepting states. The NFA description is read from a file, provided in a pre-agreed format. While processing a user input string (with the alphabet $\Sigma = \{0, 1\}$), we construct a tree of scenarios representing all possible state transitions. The NFA accepts or rejects the input string based on whether it reaches an accepting state. Our solution comprises methods for reading and processing the provided NFA and a Node class for constructing a tree of scenarios (please note that comments are removed in the following code snippets, however appear in the 'Full Code' section at the end of the document).

main: The entry point of the program (click [here](#) for explanation of line 4).

```

1 public static void main(String[] args) throws FileNotFoundException {
2     String w;
3     Scanner input = new Scanner(System.in);
4     w = getValidString();
5     simulateNFA("Figure-1-36.txt", w);
6 }

```

Figure-1-36.txt

```

3      3 states
1      1 = accepting state
1 1 2    from state 1, with symbol 1, move to 2
1 -1 3   from state 1, epsilon transition to 3
2 0 2    from state 2, with symbol 0, move to 2
2 0 3    from state 2, with symbol 0, move to 3
2 1 3    from state 2, with symbol 1, move to 3
3 0 1    from state 3, with symbol 0, move to 1

```

simulateNFA: This method takes in a string w from the user and the file "Figure-1-36.txt" containing the NFA description per the assignment guidelines displayed above. It constructs a tree of scenarios and explores all possible paths in the NFA using a breadth-first search (BFS) to determine if the input string reaches an accepting state. The method then outputs whether the input string is accepted or rejected.

Code Overview:

- read file
- initialize array of ArrayLists for epsilon transitions
- set current state to starting state
- initialize queues for scenario tree
- check epsilon transitions before first symbol
- iterate through input string and update scenario tree
 - check epsilon transitions at each tree level
 - add nodes for matching transitions
- BFS to check for accepting states after reading string & print result (ACCEPTED or REJECTED)

```

1 public static void simulateNFA(String filename, String w) throws FileNotFoundException {
2     int[] acceptingStates = new int[20];
3     int numStates = readFileDFA(filename, acceptingStates);
4     ArrayList<Integer> stateFrom = new ArrayList<>();
5     ArrayList<Integer> transition = new ArrayList<>();
6     ArrayList<Integer> stateTo = new ArrayList<>();
7     readTransitions(filename, stateFrom, transition, stateTo, numStates);
8     ArrayList<Integer>[] epsilonTransitions = new ArrayList[numStates + 1];
9
10    for (int i = 0; i < numStates + 1; i++) {
11        epsilonTransitions[i] = new ArrayList<Integer>();
12    }
13    for (int i = 0; i < transition.size(); i++) {
14        if (transition.get(i) == -1) {
15            epsilonTransitions[stateFrom.get(i)].add(stateTo.get(i));
16        }
17    }
18
19    int currentState = 1;
20    char transitionSymbol;
21    Queue<Node> nodeQueue = new LinkedList();
22    Queue<Node> nextNodeQueue = new LinkedList();
23    Node root = new Node(1);
24    nextNodeQueue.add(root);
25    Node currentNode = root;
26    Node newNode;
27
28    if (!epsilonTransitions[currentState].isEmpty()) {
29        for (int k = 0; k < epsilonTransitions[currentState].size(); k++) {
30            newNode = currentNode.insert(epsilonTransitions[currentState].get(k));
31            nextNodeQueue.add(newNode);
32        }
33    }
34
35    for (int i = 0; i < w.length(); i++) {
36        nodeQueue = new LinkedList<>(nextNodeQueue);
37        while (!nextNodeQueue.isEmpty()) {
38            nextNodeQueue.remove();
39        }
40
41        while (!nodeQueue.isEmpty()) {
42            currentNode = nodeQueue.remove();
43            currentState = (int) currentNode.getData();
44
45            if (i != 0) {
46                if (!epsilonTransitions[currentState].isEmpty()) {
47                    for (int k = 0; k < epsilonTransitions[currentState].size(); k++) {
48                        newNode = currentNode.insert(epsilonTransitions[currentState].get(k));
49                        nodeQueue.add(newNode);
50                    }
51                }
52            }
53
54            for (int j = 0; j < transition.size(); j++) {
55                if (transition.get(j) == 0) transitionSymbol = '0'; else if (
56                    transition.get(j) == 1
57                ) transitionSymbol = '1'; else transitionSymbol = 'e';
58                if (
59                    currentState == stateFrom.get(j) && w.charAt(i) == transitionSymbol
60                ) {
61                    newNode = currentNode.insert(stateTo.get(j));
62                    nextNodeQueue.add(newNode);
63                }
64            }
65        }
66    }
67    System.out.println();
68    System.out.print("BFS for tree of scenarios: ");
69    root.BFS();
70    System.out.println();
71    System.out.println("\nString " + w + " is: ");
72    if (root.acceptedBFS(acceptingStates))
73        System.out.println("ACCEPTED");
74    else System.out.println("REJECTED");
75    System.out.println();
76 }

```

readFileDFA: Reads the NFA description from the given file and returns the number of states in the NFA. Also populates the given accepting[] array with the accepting states of the NFA.

```

1  public static int readFileDFA(String filename, int accepting[])
2      throws FileNotFoundException {
3      int numStates = 0;
4      int length = -1;
5      BufferedReader in = new BufferedReader(new FileReader(filename));
6      String oneLine;
7      StringTokenizer str;
8      try {
9          oneLine = in.readLine();
10         str = new StringTokenizer(oneLine);
11         numStates = Integer.parseInt(str.nextToken());
12         oneLine = in.readLine();
13         str = new StringTokenizer(oneLine);
14         length = str.countTokens();
15         for (int i = 0; i < length; i++) {
16             accepting[i] = Integer.parseInt(str.nextToken());
17         }
18     } catch (IOException e) {
19         System.err.println("Unexpected error");
20     }
21     return numStates;
22 }

```

readTransitions: Reads the state transition information from the file with the given filename. Populates the ArrayLists 'stateFrom', 'transition', and 'stateTo' with the read transition information, enabling the simulation to know the possible transitions for each state in the NFA.

```

1  public static void readTransitions(String filename, ArrayList<Integer> from, ArrayList<Integer> transition,
2      ArrayList<Integer> to, int numStates) throws FileNotFoundException {
3      Scanner in = new Scanner(new File(filename));
4      in.nextLine();
5      in.nextLine();
6      while (in.hasNextInt()) {
7          from.add(in.nextInt());
8          transition.add(in.nextInt());
9          to.add(in.nextInt());
10     }
11 }

```

getValidString: Reads a valid input string from the user, must only contain characters '0' and '1'. Returns the read string after checking that the string is valid, ensuring that the input string adheres to this expected input alphabet.

```

1  public static String getValidString() {
2      String w;
3      boolean isBinary = false;
4      Scanner input = new Scanner(System.in);
5      System.out.print("\nEnter string in alphabet {0,1}: ");
6      w = input.nextLine();
7      while (!isBinary) {
8          for (int i = 0; i < w.length(); i++) {
9              if (w.charAt(i) != '1' && w.charAt(i) != '0') {
10                 isBinary = false;
11                 System.out.print("\nINVALID STRING. Must be in alphabet {0,1}: ");
12                 w = input.nextLine();
13             } else isBinary = true;
14         }
15     }
16     return w;
17 }

```

Node Class: Serves as the basic building block for constructing the tree of scenarios used to represent the possible state transitions in the NFA. The tree enables the exploration of all possible paths the NFA could take while reading the input string. The Node class has attributes and methods for storing state values, parent and child nodes, inserting child nodes, and performing a Breadth-First Search on the tree to determine if any paths lead to an accepting state.

```

1  import java.util.*;
2
3  public class Node<T> {
4      private T data;
5      private Node<T> parent;
6      public ArrayList<Node<T>> children;
7
8      public Node() {
9          data = null;
10         parent = null;
11         children = new ArrayList<Node<T>>();
12     }
13
14     public Node(T info) {
15         data = info;
16         parent = null;
17         children = new ArrayList<Node<T>>();
18     }
19
20     public Node(T info, Node<T> p) {
21         data = info;
22         parent = p;
23         children = new ArrayList<Node<T>>();
24     }
25
26     public String toString() {return data.toString();}
27
28     public T getData() {return data;}
29
30     public Node<T> insert(T item) {
31         Node<T> child = new Node<T>(item);
32         child.parent = this;
33         this.children.add(child);
34         return child;
35     }
36
37     public void BFS() {
38         Queue<Node<T>> q = new LinkedList<Node<T>>();
39         Node<T> v;
40         q.add(this);
41         while (!q.isEmpty()) {
42             v = q.remove();
43             System.out.print(v.data + " ");
44             for (int i = 0; i < v.children.size(); i++) {
45                 q.add(v.children.get(i));
46             }
47         }
48     }
49
50     public boolean acceptedBFS(int[] acceptStates) {
51         Queue<Node<T>> q = new LinkedList<Node<T>>();
52         Node<T> v;
53         q.add(this);
54         while (!q.isEmpty()) {
55             v = q.remove();
56
57             for (int j = 0; j < acceptStates.length; j++) {
58                 if (
59                     v.children.isEmpty() && (int) v.data == acceptStates[j]
60                 ) return true;
61             }
62
63             for (int i = 0; i < v.children.size(); i++) {
64                 q.add(v.children.get(i));
65             }
66         }
67         return false;
68     }
69 }

```

Output Screens for Input Strings:

Input String 100:

```
Enter string in alphabet {0,1}: 100
BFS for tree of scenarios: 1 3 2 2 3 2 3 1
String 100 is:
ACCEPTED
```

Input String 00:

```
Enter string in alphabet {0,1}: 00
BFS for tree of scenarios: 1 3 1 3 1
String 00 is:
ACCEPTED
```

Input String 10:

```
Enter string in alphabet {0,1}: 10
BFS for tree of scenarios: 1 3 2 2 3
String 10 is:
REJECTED
```

Full Code:

```

1  import java.io.*;
2  import java.io.BufferedReader;
3  import java.io.IOException;
4  import java.io.InputStreamReader;
5  import java.util.*;
6
7  // Java program that simulates an NFA to check if it accepts a given set of strings.
8  // It reads the NFA's definition and input strings from separate files
9  // and prints the results of the simulation for each input string.
10
11 public class ProgTask2_NFA {
12
13     public static void main(String[] args) throws FileNotFoundException {
14         String w;
15         Scanner input = new Scanner(System.in);
16         w = getValidString();
17         simulateNFA("Figure-1-36.txt", w);
18     }
19
20     public static void simulateNFA(String filename, String w)
21         throws FileNotFoundException {
22         // array with the accepting states
23         int[] acceptingStates = new int[20];
24         int numStates = readFileDFA(filename, acceptingStates);
25
26         // parallel array lists for transitions
27         ArrayList<Integer> stateFrom = new ArrayList<>();
28         ArrayList<Integer> transition = new ArrayList<>();
29         ArrayList<Integer> stateTo = new ArrayList<>();
30         // call to create the lists for transitions
31         readTransitions(filename, stateFrom, transition, stateTo, numStates);
32
33         // array of array lists of epsilon transitions
34         // index of array is the state
35         // the array list at each index holds all states an epsilon transition leads to
36         // from the state at index epsilonTransitions[i]
37
38         ArrayList<Integer>[] epsilonTransitions = new ArrayList[numStates + 1];
39         // use numStates + 1, index [0] will always be empty since no state q0 exists
40
41         for (int i = 0; i < numStates + 1; i++) { // initializes all array lists
42             epsilonTransitions[i] = new ArrayList<Integer>();
43         }
44         for (int i = 0; i < transition.size(); i++) {
45             if (transition.get(i) == -1) { // check for epsilon transitions
46                 // add the states that the epsilon transitions lead to for
47                 // each state in the NFA
48                 epsilonTransitions[stateFrom.get(i)].add(stateTo.get(i));
49             }
50         }
51
52         // currentState will keep track of the current state
53         int currentState = 1; // starting state is ALWAYS 1
54
55         // used to determine which symbol
56         // is required for a particular transition
57         char transitionSymbol;
58
59         // queues used to create tree of scenarios
60         Queue<Node> nodeQueue = new LinkedList();
61         Queue<Node> nextNodeQueue = new LinkedList();
62
63         // initialization to create the tree of scenarios
64         Node root = new Node(1);
65         nextNodeQueue.add(root);
66
67         // keeps track of the current node
68         // initialized to the root or start state
69         Node currentNode = root;
70         // variable to keep track of new nodes that are inserted
71         Node newNode;
72
73         // check for epsilon transitions before first symbol is read
74         if (!epsilonTransitions[currentState].isEmpty()) {
75             for (int k = 0; k < epsilonTransitions[currentState].size(); k++) {
76                 newNode = currentNode.insert(epsilonTransitions[currentState].get(k));
77                 nextNodeQueue.add(newNode);

```

```

78     }
79 }
80
81 // iterates through the length of the string w
82 for (int i = 0; i < w.length(); i++) {
83     // set the queue equal to the next level
84     // of nodes to be explored before clearing
85     nodeQueue = new LinkedList<>(nextNodeQueue);
86
87     // empty the queue that will store added nodes
88     // for the next level for the next
89     // symbol in w in the tree
90     while (!nextNodeQueue.isEmpty()) {
91         nextNodeQueue.remove();
92     }
93
94     // iterates through all nodes in the current level
95     // of the tree of scenarios
96     while (!nodeQueue.isEmpty()) {
97         // get the value from the first node in queue
98         currentNode = nodeQueue.remove();
99         // set the current state to the current node
100        currentState = (int) currentNode.getData();
101
102        // check for epsilon transitions now that the state has changed
103        if (i != 0) { // skip first iteration since epsilon transitions were checked once prior
104            if (!epsilonTransitions[currentState].isEmpty()) {
105                for (int k = 0; k < epsilonTransitions[currentState].size(); k++) {
106                    newNode =
107                        currentNode.insert(epsilonTransitions[currentState].get(k));
108                    nodeQueue.add(newNode); // add the newest node from epsilon transition to queue
109                }
110            }
111        }
112
113        // iterates through all possible transitions
114        for (int j = 0; j < transition.size(); j++) {
115            // read the required transition symbol from the parallel arrays
116            // and convert it to a char value so it can be compared to
117            // the current symbol in w
118            if (transition.get(j) == 0) transitionSymbol = '0'; else if (
119                transition.get(j) == 1
120            ) transitionSymbol = '1'; else transitionSymbol = 'e';
121
122            // check if the current state matches the first state
123            // in the parallel transition arrays and if the current symbol
124            // in the string matches the required symbol for a transition
125            // to a new state
126            if (
127                currentState == stateFrom.get(j) && w.charAt(i) == transitionSymbol
128            ) {
129                // if requirements are met then add to tree of scenarios
130                newNode = currentNode.insert(stateTo.get(j));
131                // add the new node to the queue for the next level of the tree
132                nextNodeQueue.add(newNode);
133            }
134        }
135    }
136 }
137
138 // DEBUGGING
139 System.out.println();
140 System.out.print("BFS for tree of scenarios: ");
141 root.BFS();
142 System.out.println();
143
144 // print results
145 System.out.println("\nString " + w + " is: ");
146 if (root.acceptedBFS(acceptingStates)) System.out.println(
147     "ACCEPTED"
148 ); else System.out.println("REJECTED");
149 System.out.println();
150 }
151
152 public static int readFileDFA(String filename, int accepting[])
153     throws FileNotFoundException {
154     int numStates = 0;
155     int length = -1;
156     BufferedReader in = new BufferedReader(new FileReader(filename));
157     String oneLine;

```

```

158 StringTokenizer str;
159 // get the number of states and
160 // get the accepting states and store in array
161 try {
162     oneLine = in.readLine();
163     str = new StringTokenizer(oneLine);
164     numStates = Integer.parseInt(str.nextToken());
165     oneLine = in.readLine();
166     str = new StringTokenizer(oneLine);
167     length = str.countTokens();
168     for (int i = 0; i < length; i++) {
169         accepting[i] = Integer.parseInt(str.nextToken());
170     }
171 } catch (IOException e) {
172     System.err.println("Unexpected error");
173 }
174 return numStates;
175 }
176
177 public static void readTransitions(
178     String filename,
179     ArrayList<Integer> from,
180     ArrayList<Integer> transition,
181     ArrayList<Integer> to,
182     int numStates
183 ) throws FileNotFoundException {
184     Scanner in = new Scanner(new File(filename));
185     in.nextLine(); // discard number of states in file
186     in.nextLine(); // discard transitions in file
187     // use 3 parallel arrays
188     while (in.hasNextInt()) {
189         from.add(in.nextInt());
190         transition.add(in.nextInt());
191         to.add(in.nextInt());
192     }
193 }
194
195 // method that gets a valid string
196 public static String getValidString() {
197     String w;
198     boolean isBinary = false;
199     Scanner input = new Scanner(System.in);
200     System.out.print("\nEnter string in alphabet {0,1}: ");
201     w = input.nextLine();
202     // loops until a string with only 0's and 1's is input
203     while (!isBinary) {
204         for (int i = 0; i < w.length(); i++) {
205             if (w.charAt(i) != '1' && w.charAt(i) != '0') {
206                 isBinary = false;
207                 System.out.print("\nINVALID STRING. Must be in alphabet {0,1}: ");
208                 w = input.nextLine();
209             } else isBinary = true;
210         }
211     }
212     return w;
213 }
214 }
215
216 // Node class for tree of scenarios
217 import java.util.*;
218
219 public class Node<T> {
220
221     private T data;
222     private Node<T> parent;
223     public ArrayList<Node<T>> children;
224
225     // default constructor
226     public Node() {
227         data = null;
228         parent = null;
229         children = new ArrayList<Node<T>>();
230     }
231
232     // alternate constructor
233     public Node(T info) {
234         data = info;
235         parent = null;
236         children = new ArrayList<Node<T>>();
237     }

```



```

238
239 // alternate constructor
240 public Node(T info, Node<T> p) {
241     data = info;
242     parent = p;
243     children = new ArrayList<Node<T>>();
244 }
245
246 // toString
247 public String toString() {
248     return data.toString();
249 }
250
251 // returns state value of the node
252 public T getData() {
253     return data;
254 }
255
256 // adds a child to tree of scenarios
257 public Node<T> insert(T item) {
258     Node<T> child = new Node<T>(item);
259     child.parent = this;
260     this.children.add(child);
261     return child;
262 }
263
264 // BFS that prints the state values of the nodes
265 public void BFS() {
266     Queue<Node<T>> q = new LinkedList<Node<T>>();
267     Node<T> v;
268     q.add(this);
269     while (!q.isEmpty()) {
270         v = q.remove();
271         System.out.print(v.data + " ");
272         for (int i = 0; i < v.children.size(); i++) {
273             q.add(v.children.get(i));
274         }
275     }
276 }
277
278 // BFS that checks if any leaf nodes are accepting
279 public boolean acceptedBFS(int[] acceptStates) {
280     Queue<Node<T>> q = new LinkedList<Node<T>>();
281     Node<T> v;
282     q.add(this);
283     while (!q.isEmpty()) {
284         v = q.remove();
285
286         // iterate through all possible accept states
287         for (int j = 0; j < acceptStates.length; j++) {
288             if (
289                 v.children.isEmpty() && (int) v.data == acceptStates[j]
290             ) return true;
291         }
292
293         for (int i = 0; i < v.children.size(); i++) {
294             q.add(v.children.get(i));
295         }
296     }
297
298     return false;
299 }
300 }

```