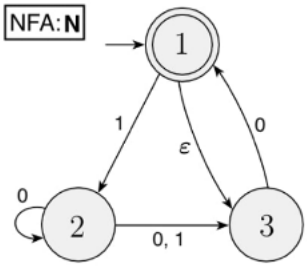
Assignment Summary:



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

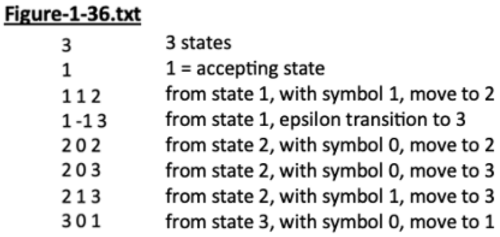


Figure-1-36.txt

```
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)

```java
public static void simulateNFA(String filename, String w) throws FileNotFoundException {
  int[] acceptingStates = new int[20];
  int numStates = readFileDFA(filename, acceptingStates);
  ArrayList<Integer> stateFrom = new ArrayList<>();
  ArrayList<Integer> transition = new ArrayList<>();
  ArrayList<Integer> stateTo = new ArrayList<>();
  readTransitions(filename, stateFrom, transition, stateTo, numStates);
  ArrayList<Integer>[] epsilonTransitions = new ArrayList[numStates + 1];

  for (int i = 0; i < numStates + 1; i++) {
    epsilonTransitions[i] = new ArrayList<Integer>();
  }
  for (int i = 0; i < transition.size(); i++) {
    if (transition.get(i) == -1) {
      epsilonTransitions[stateFrom.get(i)].add(stateTo.get(i));
    }
  }

  int currentState = 1;
  char transitionSymbol;
  Queue<Node> nodeQueue = new LinkedList();
  Queue<Node> nextNodeQueue = new LinkedList();
  Node root = new Node(1);
  nextNodeQueue.add(root);
  Node currentNode = root;
  Node newNode;

  if (!epsilonTransitions[currentState].isEmpty()) {
    for (int k = 0; k < epsilonTransitions[currentState].size(); k++) {
      newNode = currentNode.insert(epsilonTransitions[currentState].get(k));
      nextNodeQueue.add(newNode);
    }
  }

  for (int i = 0; i < w.length(); i++) {
    nodeQueue = new LinkedList<>(nextNodeQueue);
    while (!nextNodeQueue.isEmpty()) {
      nextNodeQueue.remove();
    }

    while (!nodeQueue.isEmpty()) {
      currentNode = nodeQueue.remove();
      currentState = (int) currentNode.getData();

      if (i != 0) {
        if (!epsilonTransitions[currentState].isEmpty()) {
          for (int k = 0; k < epsilonTransitions[currentState].size(); k++) {
            newNode = currentNode.insert(epsilonTransitions[currentState].get(k));
            nodeQueue.add(newNode);
          }
        }
      }

      for (int j = 0; j < transition.size(); j++) {
        if (transition.get(j) == 0) transitionSymbol = '0'; else if (
          transition.get(j) == 1
        ) transitionSymbol = '1'; else transitionSymbol = 'e';
        if (
          currentState == stateFrom.get(j) && w.charAt(i) == transitionSymbol
        ) {
          newNode = currentNode.insert(stateTo.get(j));
          nextNodeQueue.add(newNode);
        }
      }
    }
  }
  System.out.println();
  System.out.print("BFS for tree of scenarios: ");
  root.BFS();
  System.out.println();
  System.out.println("\nString " + w + " is: ");
  if (root.acceptedBFS(acceptingStates))
      System.out.println("ACCEPTED");
  else System.out.println("REJECTED");
      System.out.println();
}
```

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.

```java
public static int readFileDFA(String filename, int accepting[])
  throws FileNotFoundException {
  int numStates = 0;
  int length = -1;
  BufferedReader in = new BufferedReader(new FileReader(filename));
  String oneLine;
  StringTokenizer str;
  try {
    oneLine = in.readLine();
    str = new StringTokenizer(oneLine);
    numStates = Integer.parseInt(str.nextToken());
    oneLine = in.readLine();
    str = new StringTokenizer(oneLine);
    length = str.countTokens();
    for (int i = 0; i < length; i++) {
      accepting[i] = Integer.parseInt(str.nextToken());
    }
  } catch (IOException e) {
    System.err.println("Unexpected error");
  }
  return numStates;
}
```

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.

```java
public static void readTransitions(String filename, ArrayList<Integer> from, ArrayList<Integer> transition,
ArrayList<Integer> to, int numStates) throws FileNotFoundException {
  Scanner in = new Scanner(new File(filename));
  in.nextLine();
  in.nextLine();
  while (in.hasNextInt()) {
    from.add(in.nextInt());
    transition.add(in.nextInt());
    to.add(in.nextInt());
  }
}
```

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.

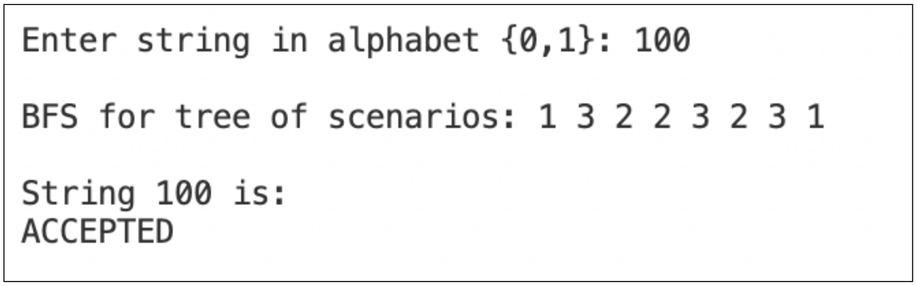
```java
public static String getValidString() {
  String w;
  boolean isBinary = false;
  Scanner input = new Scanner(System.in);
  System.out.print("\nEnter string in alphabet {0,1}: ");
  w = input.nextLine();
  while (!isBinary) {
    for (int i = 0; i < w.length(); i++) {
      if (w.charAt(i) != '1' && w.charAt(i) != '0') {
        isBinary = false;
        System.out.print("\nINVALID STRING. Must be in alphabet {0,1}: ");
        w = input.nextLine();
      } else isBinary = true;
    }
  }
  return w;
}
```

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.

```java
import java.util.*;

public class Node<T> {
  private T data;
  private Node<T> parent;
  public ArrayList<Node<T>> children;

  public Node() {
    data = null;
    parent = null;
    children = new ArrayList<Node<T>>();
  }

  public Node(T info) {
    data = info;
    parent = null;
    children = new ArrayList<Node<T>>();
  }

  public Node(T info, Node<T> p) {
    data = info;
    parent = p;
    children = new ArrayList<Node<T>>();
  }

  public String toString() {return data.toString();}

  public T getData() {return data;}

  public Node<T> insert(T item) {
    Node<T> child = new Node<T>(item);
    child.parent = this;
    this.children.add(child);
    return child;
  }

  public void BFS() {
    Queue<Node<T>> q = new LinkedList<Node<T>>();
    Node<T> v;
    q.add(this);
    while (!q.isEmpty()) {
      v = q.remove();
      System.out.print(v.data + " ");
      for (int i = 0; i < v.children.size(); i++) {
        q.add(v.children.get(i));
      }
    }
  }

  public boolean acceptedBFS(int[] acceptStates) {
    Queue<Node<T>> q = new LinkedList<Node<T>>();
    Node<T> v;
    q.add(this);
    while (!q.isEmpty()) {
      v = q.remove();

      for (int j = 0; j < acceptStates.length; j++) {
        if (
          v.children.isEmpty() && (int) v.data == acceptStates[j]
        ) return true;
      }

      for (int i = 0; i < v.children.size(); i++) {
        q.add(v.children.get(i));
      }
    }
    return false;
  }
}
```

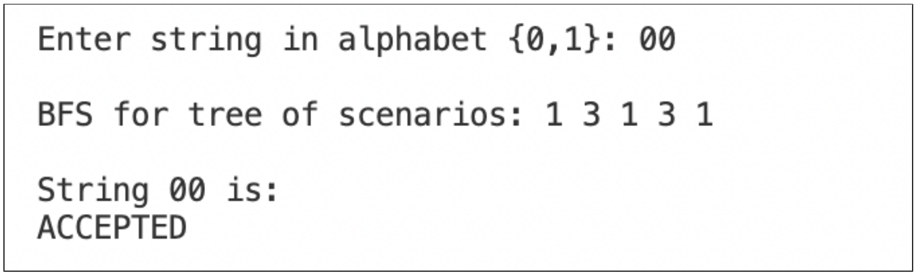
Output Screens for Input Strings:

Input String 100:



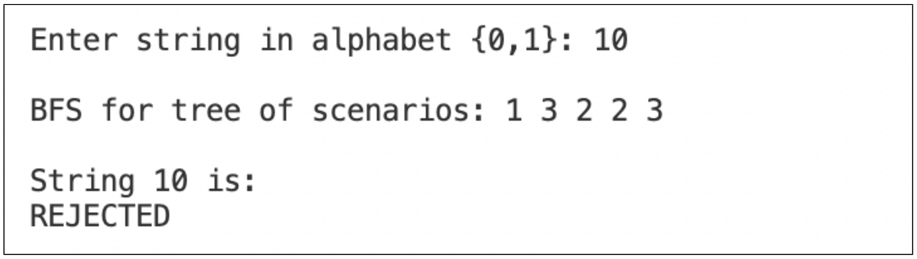
```
Enter string in alphabet {0,1}: 100

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:

```java
import java.io.*;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.*;

// Java program that simulates an NFA to check if it accepts a given set of strings.
// It reads the NFA's definition and input strings from separate files
// and prints the results of the simulation for each input string.

public class ProgTask2_NFA {

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

  public static void simulateNFA(String filename, String w)
    throws FileNotFoundException {
    // array with the accepting states
    int[] acceptingStates = new int[20];
    int numStates = readFileDFA(filename, acceptingStates);

    // parallel array lists for transitions
    ArrayList<Integer> stateFrom = new ArrayList<>();
    ArrayList<Integer> transition = new ArrayList<>();
    ArrayList<Integer> stateTo = new ArrayList<>();
    // call to create the lists for transitions
    readTransitions(filename, stateFrom, transition, stateTo, numStates);

    // array of array lists of epsilon transitions
    // index of array is the state
    // the array list at each index holds all states an epsilon transition leads to
    // from the state at index epsilonTransitions[i]

    ArrayList<Integer>[] epsilonTransitions = new ArrayList[numStates + 1];
    // use numStates + 1, index [0] will always be empty since no state q0 exists

    for (int i = 0; i < numStates + 1; i++) { // initializes all array lists
      epsilonTransitions[i] = new ArrayList<Integer>();
    }
    for (int i = 0; i < transition.size(); i++) {
      if (transition.get(i) == -1) { // check for epsilon transitions
        // add the states that the epsilon transitions lead to for
        // each state in the NFA
        epsilonTransitions[stateFrom.get(i)].add(stateTo.get(i));
      }
    }

    // currentState will keep track of the current state
    int currentState = 1; // starting state is ALWAYS 1

    // used to determine which symbol
    // is requiered for a particular transition
    char transitionSymbol;

    // queues used to create tree of scenarios
    Queue<Node> nodeQueue = new LinkedList();
    Queue<Node> nextNodeQueue = new LinkedList();

    // initialization to create the tree of scenarios
    Node root = new Node(1);
    nextNodeQueue.add(root);

    // keeps track of the current node
    // initialized to the root or start state
    Node currentNode = root;
    // variable to keep track of new nodes that are inserted
    Node newNode;

    // check for epsilon transitions before first symbol is read
    if (!epsilonTransitions[currentState].isEmpty()) {
      for (int k = 0; k < epsilonTransitions[currentState].size(); k++) {
        newNode = currentNode.insert(epsilonTransitions[currentState].get(k));
        nextNodeQueue.add(newNode);
      }
    }

    // iterates through the length of the string w
    for (int i = 0; i < w.length(); i++) {
      // set the queue equal to the next level
      // of nodes to be explored before clearing
      nodeQueue = new LinkedList<>(nextNodeQueue);

      // empty the queue that will store added nodes
      // for the next level for the next
      // symbol in w in the tree
      while (!nextNodeQueue.isEmpty()) {
        nextNodeQueue.remove();
      }

      // iterates through all nodes in the current level
      // of the tree of senarios
      while (!nodeQueue.isEmpty()) {
        // get the value from the first node in queue
        currentNode = nodeQueue.remove();
        // set the current state to the current node
        currentState = (int) currentNode.getData();

        // check for epsilon transitions now that the state has changed
        if (i != 0) { // skip first iteration since epsilon transitions were checked once prior
          if (!epsilonTransitions[currentState].isEmpty()) {
            for (int k = 0; k < epsilonTransitions[currentState].size(); k++) {
              newNode =
                currentNode.insert(epsilonTransitions[currentState].get(k));
              nodeQueue.add(newNode); // add the newest node from epsilon transition to queue
            }
          }
        }

        // iterates through all possible transitions
        for (int j = 0; j < transition.size(); j++) {
          // read the required transition symbol from the parallel arrays
          // and convert it to a char value so it can be compared to
          // the current symbol in w
          if (transition.get(j) == 0) transitionSymbol = '0'; else if (
            transition.get(j) == 1
          ) transitionSymbol = '1'; else transitionSymbol = 'e';

          // check if the current state matches the first state
          // in the parallel transition arrays and if the current symbol
          // in the string matches the required symbol for a transition
          // to a new state
          if (
            currentState == stateFrom.get(j) && w.charAt(i) == transitionSymbol
          ) {
            // if requirements are met then add to tree of scenarios
            newNode = currentNode.insert(stateTo.get(j));
            // add the new node to the queue for the next level of the tree
            nextNodeQueue.add(newNode);
          }
        }
      }
    }

    // DEBUGGING
    System.out.println();
    System.out.print("BFS for tree of scenarios: ");
    root.BFS();
    System.out.println();

    // print results
    System.out.println("\nString " + w + " is: ");
    if (root.acceptedBFS(acceptingStates)) System.out.println(
      "ACCEPTED"
    ); else System.out.println("REJECTED");
    System.out.println();
  }

  public static int readFileDFA(String filename, int accepting[])
    throws FileNotFoundException {
    int numStates = 0;
    int length = -1;
    BufferedReader in = new BufferedReader(new FileReader(filename));
    String oneLine;
    StringTokenizer str;
    // get the mumber of states and
    // get the accepting states and store in array
    try {
      oneLine = in.readLine();
      str = new StringTokenizer(oneLine);
      numStates = Integer.parseInt(str.nextToken());
      oneLine = in.readLine();
      str = new StringTokenizer(oneLine);
      length = str.countTokens();
      for (int i = 0; i < length; i++) {
        accepting[i] = Integer.parseInt(str.nextToken());
      }
    } catch (IOException e) {
      System.err.println("Unexpected error");
    }
    return numStates;
  }

  public static void readTransitions(
    String filename,
    ArrayList<Integer> from,
    ArrayList<Integer> transition,
    ArrayList<Integer> to,
    int numStates
  ) throws FileNotFoundException {
    Scanner in = new Scanner(new File(filename));
    in.nextLine(); // dicard number of states in file
    in.nextLine(); // discard transitions in file
    // use 3 parallel arrays
    while (in.hasNextInt()) {
      from.add(in.nextInt());
      transition.add(in.nextInt());
      to.add(in.nextInt());
    }
  }

  // method that gets a valid string
  public static String getValidString() {
    String w;
    boolean isBinary = false;
    Scanner input = new Scanner(System.in);
    System.out.print("\nEnter string in alphabet {0,1}: ");
    w = input.nextLine();
    // loops until a string with only 0's and 1's is input
    while (!isBinary) {
      for (int i = 0; i < w.length(); i++) {
        if (w.charAt(i) != '1' && w.charAt(i) != '0') {
          isBinary = false;
          System.out.print("\nINVALID STRING. Must be in alphabet {0,1}: ");
          w = input.nextLine();
        } else isBinary = true;
      }
    }
    return w;
  }
}

// Node class for tree of scenarios
import java.util.*;

public class Node<T> {

  private T data;
  private Node<T> parent;
  public ArrayList<Node<T>> children;

  // default constructor
  public Node() {
    data = null;
    parent = null;
    children = new ArrayList<Node<T>>();
  }

  // alternate constructor
  public Node(T info) {
    data = info;
    parent = null;
    children = new ArrayList<Node<T>>();
  }
```

```java

  // alternate constructor
  public Node(T info, Node<T> p) {
    data = info;
    parent = p;
    children = new ArrayList<Node<T>>();
  }

  // toString
  public String toString() {
    return data.toString();
  }

  // returns state value of the node
  public T getData() {
    return data;
  }

  // adds a child to tree of scenarios
  public Node<T> insert(T item) {
    Node<T> child = new Node<T>(item);
    child.parent = this;
    this.children.add(child);
    return child;
  }

  // BFS that prints the state values of the nodes
  public void BFS() {
    Queue<Node<T>> q = new LinkedList<Node<T>>();
    Node<T> v;
    q.add(this);
    while (!q.isEmpty()) {
      v = q.remove();
      System.out.print(v.data + " ");
      for (int i = 0; i < v.children.size(); i++) {
        q.add(v.children.get(i));
      }
    }
  }

  // BFS that checks if any leaf nodes are accepting
  public boolean acceptedBFS(int[] acceptStates) {
    Queue<Node<T>> q = new LinkedList<Node<T>>();
    Node<T> v;
    q.add(this);
    while (!q.isEmpty()) {
      v = q.remove();

      // iterate through all possible accept states
      for (int j = 0; j < acceptStates.length; j++) {
        if (
          v.children.isEmpty() && (int) v.data == acceptStates[j]
        ) return true;
      }

      for (int i = 0; i < v.children.size(); i++) {
        q.add(v.children.get(i));
      }
    }

    return false;
  }
}
```