**PART 1**

import java.io.IOException;

import java.nio.charset.StandardCharsets;

import java.nio.file.Files;

import java.nio.file.Paths;

import java.util.ArrayList;

import java.util.HashMap;

import java.util.LinkedList;

import java.util.Queue;

import java.util.Locale;

import java.text.DecimalFormat;

import java.text.DecimalFormatSymbols;

**public class Part1** {

static HashMap<String, Node> masterMap;

**static class Edge** {

String beforePos;

Node node;

double probability;

public Edge(String beforePos, Node node, double probability) {

this.beforePos = beforePos;

this.node = node;

this.probability = probability;

}

public String getBeforePos() { return this.beforePos; }

public Node getNode() { return this.node; }

public double getProbability() { return this.probability; }

}

**static class Node** {

String word;

HashMap<String, ArrayList<Edge>> posMap;

public Node(String word, ArrayList<String> sentenceSpec) {

this.word = word;

this.posMap = new HashMap<String, ArrayList<Edge>>();

for (String pos : sentenceSpec) {

posMap.put(pos, new ArrayList<Edge>());

}

}

public String getWord() { return this.word; }

public ArrayList<Edge> getEdges(String pos) { return posMap.get(pos); }

public void addEdge(String pos1, String pos2, Node node, double probability) {

Edge edge = new Edge(pos1, node, probability);

posMap.get(pos2).add(edge);

}

}

**static class Word** {

String text;

String pos;

double probability;

public Word(String text, String pos, double probability) {

this.text = text;

this.pos = pos;

this.probability = probability;

}

public String getText() { return this.text; }

public String getPos() { return this.pos; }

public double getProbability() { return this.probability; }

}

**static class Sequence** {

ArrayList<Word> words;

public Sequence() {

this.words = new ArrayList<Word>();

}

private Sequence(ArrayList<Word> words) {

this.words = new ArrayList<Word>(words);

}

public Sequence copy() {

return new Sequence(this.words);

}

public int size() {

return words.size();

}

public Word getLastWord() {

return words.get(words.size() - 1);

}

public String getSentence(){

ArrayList<String> texts = new ArrayList<String>();

for (Word word : words) {

texts.add(word.getText());

}

return String.join(" ", texts);

}

public double getTotalProbability() {

double probability = 1;

for (Word word : words) {

probability \*= word.getProbability();

}

return probability;

}

public boolean addWord(String word, String pos, double probability, ArrayList<String> sentenceSpec) {

if (words.size() < sentenceSpec.size() && sentenceSpec.get(words.size()).equals(pos)) {

this.words.add(new Word(word, pos, probability));

return true;

}

return false;

}

}

**private static Node getNodeFromMaster**(String word, ArrayList<String> sentenceSpec) {

if (masterMap.get(word) == null) {

Node newNode = new Node(word, sentenceSpec);

masterMap.put(word, newNode);

return newNode;

}

else { return masterMap.get(word); }

}

**public static String bfs**(Node root, ArrayList<String> sentenceSpec) {

ArrayList<Sequence> validSequences = new ArrayList<Sequence>();

int nodesConsidered = 0;

Sequence rootSeq = new Sequence();

rootSeq.addWord(root.getWord(), sentenceSpec.get(0), 1, sentenceSpec);

Queue queue = new LinkedList();

queue.add(rootSeq);

while(!queue.isEmpty()) {

Sequence seq = (Sequence)queue.remove();

if (seq.size() == sentenceSpec.size()) {

validSequences.add(seq);

continue;

}

Word word = seq.getLastWord();

Node node = masterMap.get(word.getText());

String nextPos = sentenceSpec.get(seq.size());

for (Edge edge : node.getEdges(nextPos)) {

nodesConsidered++;

if (edge.getBeforePos().equals(word.getPos())) {

Sequence newSeq = seq.copy();

Node nextNode = edge.getNode();

if (newSeq.addWord(nextNode.getWord(), nextPos, edge.getProbability(), sentenceSpec)){

queue.add(newSeq);

}

}

}

}

double maxProbability = 0;

Sequence maxProbabilitySeq = null;

for (Sequence seq : validSequences){

if (seq.getTotalProbability() > maxProbability) {

maxProbability = seq.getTotalProbability();

maxProbabilitySeq = seq;

}

}

DecimalFormat df = new DecimalFormat("0", DecimalFormatSymbols.getInstance(Locale.ENGLISH));

df.setMaximumFractionDigits(340);

return "\"" + maxProbabilitySeq.getSentence() + "\" with probability " + df.format(maxProbability) + "\nTotal nodes considered: " + nodesConsidered;

}

**public static String generate**(String startingWord, ArrayList<String> sentenceSpec, String graph) {

masterMap = new HashMap<String, Node>();

// Parse input

for (String line : graph.split("\n")) {

String[] parts = line.split("//");

String[] firstTag = parts[0].split("/");

String[] secondTag = parts[1].split("/");

double probability = Double.parseDouble(parts[2]);

Node n1 = getNodeFromMaster(firstTag[0], sentenceSpec);

Node n2 = getNodeFromMaster(secondTag[0], sentenceSpec);

String pos1 = firstTag[1];

String pos2 = secondTag[1];

if (sentenceSpec.contains(pos1) && sentenceSpec.contains(pos2)) {

n1.addEdge(pos1, pos2, n2, probability);

}

}

return bfs(getNodeFromMaster(startingWord, sentenceSpec), sentenceSpec);

}

}

**PART 2**

1) 

2) 

3) 

4) 

**PART 3**

import java.io.IOException;

import java.nio.charset.StandardCharsets;

import java.nio.file.Files;

import java.nio.file.Paths;

import java.util.ArrayList;

import java.util.HashMap;

import java.util.LinkedList;

import java.util.Queue;

import java.util.Stack;

import java.util.Locale;

import java.text.DecimalFormat;

import java.text.DecimalFormatSymbols;

**public class Part3** {

static HashMap<String, Node> masterMap;

**class Edge** {

String beforePos;

Node node;

double probability;

public Edge(String beforePos, Node node, double probability) {

this.beforePos = beforePos;

this.node = node;

this.probability = probability;

}

public String getBeforePos() { return this.beforePos; }

public Node getNode() { return this.node; }

public double getProbability() { return this.probability; }

}

**static class Node** {

String word;

HashMap<String, ArrayList<Edge>> posMap;

public Node(String word, ArrayList<String> sentenceSpec) {

this.word = word;

this.posMap = new HashMap<String, ArrayList<Edge>>();

for (String pos : sentenceSpec) {

posMap.put(pos, new ArrayList<Edge>());

}

}

public String getWord() { return this.word; }

public ArrayList<Edge> getEdges(String pos) { return posMap.get(pos); }

public void addEdge(String pos1, String pos2, Node node, double probability) {

Edge edge = new Edge(pos1, node, probability);

posMap.get(pos2).add(edge);

}

}

**static class Word** {

String text;

String pos;

double probability;

public Word(String text, String pos, double probability) {

this.text = text;

this.pos = pos;

this.probability = probability;

}

public String getText() { return this.text; }

public String getPos() { return this.pos; }

public double getProbability() { return this.probability; }

}

**static class Child** {

Word word;

boolean visited;

public Child(Word word) {

this.word = word;

this.visited = false;

}

public Word getWord() { this.word; }

public boolean isVisited() { this.visited; }

public void visit() { this.visited = true; }

}

**static class Sequence** {

ArrayList<Word> words;

ArrayList<Child> children;

public Sequence() {

this.words = new ArrayList<Word>();

this.children = new ArrayList<Child>();

}

private Sequence(ArrayList<Word> words) {

this.words = new ArrayList<Word>(words);

this.children = new ArrayList<Child>();

}

public Sequence copy() {

return new Sequence(this.words);

}

public int size() {

return words.size();

}

public Word getLastWord() {

return words.get(words.size() - 1);

}

public String getSentence() {

ArrayList<String> texts = new ArrayList<String>();

for (Word word : words) {

texts.add(word.getText());

}

return String.join(" ", texts);

}

public double getTotalProbability() {

double probability = 1;

for (Word word : words) {

probability \*= word.getProbability();

}

return probability;

}

public Word getNextChild() {

for (Child child : children) {

if (!child.isVisited()) {

child.visit();

return child.getWord();

}

}

return null;

}

public boolean hasChildren() {

return !this.children.isEmpty();

}

public boolean addWord(String word, String pos, double probability, ArrayList<String> sentenceSpec) {

if (words.size() < sentenceSpec.size() && sentenceSpec.get(words.size()).equals(pos)) {

this.words.add(new Word(word, pos, probability));

return true;

}

return false;

}

public boolean addWord(Word word, ArrayList<String> sentenceSpec) {

if (words.size() < sentenceSpec.size() && sentenceSpec.get(words.size()).equals(word.getPos())) {

this.words.add(word);

return true;

}

return false;

}

public void addChild(String word, String pos, double probability, ArrayList<String> sentenceSpec){

if (words.size() < sentenceSpec.size() && sentenceSpec.get(words.size()).equals(pos)) {

Word childWord = new Word(word, pos, probability);

children.add(new Child(childWord));

}

}

}

**private static Node getNodeFromMaster**(String word, ArrayList<String> sentenceSpec) {

if (masterMap.get(word) == null) {

Node newNode = new Node(word, sentenceSpec);

masterMap.put(word, newNode);

return newNode;

}

else { return masterMap.get(word); }

}

// BREADTH FIRST SEARCH

**public static String bfs**(Node root, ArrayList<String> sentenceSpec) {

ArrayList<Sequence> validSequences = new ArrayList<Sequence>();

int nodesConsidered = 0;

Sequence rootSeq = new Sequence();

rootSeq.addWord(root.getWord(), sentenceSpec.get(0), 1, sentenceSpec);

Queue queue = new LinkedList();

queue.add(rootSeq);

while(!queue.isEmpty()) {

Sequence seq = (Sequence)queue.remove();

if (seq.size() == sentenceSpec.size()) {

validSequences.add(seq);

continue;

}

Word word = seq.getLastWord();

Node node = masterMap.get(word.getText());

String nextPos = sentenceSpec.get(seq.size());

for (Edge edge : node.getEdges(nextPos)) {

nodesConsidered++;

if (edge.getBeforePos().equals(word.getPos())) {

Sequence newSeq = seq.copy();

Node nextNode = edge.getNode();

if (newSeq.addWord(nextNode.getWord(), nextPos, edge.getProbability(), sentenceSpec)){

queue.add(newSeq);

}

}

}

}

double maxProbability = 0;

Sequence maxProbabilitySeq = null;

for (Sequence seq : validSequences){

if (seq.getTotalProbability() > maxProbability) {

maxProbability = seq.getTotalProbability();

maxProbabilitySeq = seq;

}

}

DecimalFormat df = new DecimalFormat("0", DecimalFormatSymbols.getInstance(Locale.ENGLISH));

df.setMaximumFractionDigits(340);

return "\"" + maxProbabilitySeq.getSentence() + "\" with probability " + df.format(maxProbability) + "\nTotal nodes considered: " + nodesConsidered;

}

// DEPTH FIRST SEARCH

**public static String dfs**(Node root, ArrayList<String> sentenceSpec) {

ArrayList<Sequence> validSequences = new ArrayList<Sequence>();

int nodesConsidered = 0;

Sequence rootSeq = new Sequence();

rootSeq.addWord(root.getWord(), sentenceSpec.get(0), 1, sentenceSpec);

Stack stack = new Stack();

stack.push(rootSeq);

while(!stack.isEmpty()) {

Sequence seq = (Sequence)stack.peek();

if (seq.size() == sentenceSpec.size()) {

validSequences.add(seq);

stack.pop();

continue;

}

if (!seq.hasChildren()) {

Word word = seq.getLastWord();

Node node = masterMap.get(word.getText());

String nextPos = sentenceSpec.get(seq.size());

for (Edge edge : node.getEdges(nextPos)) {

nodesConsidered++;

if (edge.getBeforePos().equals(word.getPos())) {

Node nextNode = edge.getNode();

seq.addChild(nextNode.getWord(), nextPos, edge.getProbability(), sentenceSpec);

}

}

}

Word child = seq.getNextChild();

if (child != null) {

Sequence newSeq = seq.copy();

if (newSeq.addWord(child, sentenceSpec)){

stack.push(newSeq);

}

}

else {

stack.pop();

}

}

double maxProbability = 0;

Sequence maxProbabilitySeq = null;

for (Sequence seq : validSequences){

if (seq.getTotalProbability() > maxProbability) {

maxProbability = seq.getTotalProbability();

maxProbabilitySeq = seq;

}

}

DecimalFormat df = new DecimalFormat("0", DecimalFormatSymbols.getInstance(Locale.ENGLISH));

df.setMaximumFractionDigits(340);

return "\"" + maxProbabilitySeq.getSentence() + "\" with probability " + df.format(maxProbability) + "\nTotal nodes considered: " + nodesConsidered;

}

// HEURISTIC SEARCH

**public static String heuristic**(Node root, ArrayList<String> sentenceSpec) {

HashMap<Sequence, Double> sequences = new HashMap<Sequence, Double>();

int nodesConsidered = 0;

Sequence rootSeq = new Sequence();

rootSeq.addWord(root.getWord(), sentenceSpec.get(0), 1, sentenceSpec);

sequences.put(rootSeq, rootSeq.getTotalProbability());

Sequence maxProbabilitySeq = null;

while (true) {

double maxProbability = 0;

for (Sequence seq : sequences.keySet()) {

nodesConsidered++;

if (sequences.get(seq) > maxProbability) {

maxProbabilitySeq = seq;

}

}

sequences.remove(maxProbabilitySeq);

if (maxProbabilitySeq.size() == sentenceSpec.size()) {

break;

}

else {

Word word = maxProbabilitySeq.getLastWord();

Node node = masterMap.get(word.getText());

String nextPos = sentenceSpec.get(maxProbabilitySeq.size());

for (Edge edge : node.getEdges(nextPos)) {

if (edge.getBeforePos().equals(word.getPos())) {

Sequence newSeq = maxProbabilitySeq.copy();

Node nextNode = edge.getNode();

if (newSeq.addWord(nextNode.getWord(), nextPos, edge.getProbability(), sentenceSpec)){

sequences.put(newSeq, estimateProbability(nextNode, sentenceSpec, newSeq.size(), newSeq.getTotalProbability()));

}

}

}

}

}

DecimalFormat df = new DecimalFormat("0", DecimalFormatSymbols.getInstance(Locale.ENGLISH));

df.setMaximumFractionDigits(340);

return "\"" + maxProbabilitySeq.getSentence() + "\" with probability " + df.format(maxProbabilitySeq.getTotalProbability()) + "\nTotal nodes considered: " + nodesConsidered;

}

**private static double estimateProbability**(Node node, ArrayList<String> sentenceSpec, int index, double probability) {

if (index == sentenceSpec.size()) {

return probability;

}

else {

double newProbability = probability;

for (Edge edge : node.getEdges(sentenceSpec.get(index))) {

if (edge.getBeforePos().equals(sentenceSpec.get(index))) {

newProbability \*= estimateProbability(edge.getNode(), sentenceSpec, index + 1, edge.getProbability());

}

}

return newProbability;

}

}

**public static String generate**(String startingWord, ArrayList<String> sentenceSpec, String searchStrategy, String graph) {

masterMap = new HashMap<String, Node>();

// Parse input

for (String line : graph.split("\n")) {

String[] parts = line.split("//");

String[] firstTag = parts[0].split("/");

String[] secondTag = parts[1].split("/");

double probability = Double.parseDouble(parts[2]);

Node n1 = getNodeFromMaster(firstTag[0], sentenceSpec);

Node n2 = getNodeFromMaster(secondTag[0], sentenceSpec);

String pos1 = firstTag[1];

String pos2 = secondTag[1];

if (sentenceSpec.contains(pos1) && sentenceSpec.contains(pos2)) {

n1.addEdge(pos1, pos2, n2, probability);

}

}

if (searchStrategy == "BREADTH\_FIRST") {

return bfs(getNodeFromMaster(startingWord, sentenceSpec), sentenceSpec);

}

if (searchStrategy == "DEPTH\_FIRST") {

return dfs(getNodeFromMaster(startingWord, sentenceSpec), sentenceSpec);

}

if (searchStrategy == "HEURISTIC") {

return heuristic(getNodeFromMaster(startingWord, sentenceSpec), sentenceSpec);

}

return null;

}

}

BREADTH-FIRST

1. 
2. 
3. 
4. 

DEPTH-FIRST

1. 
2. 
3. 
4. 

HEURISTIC

1. 
2. 
3. 
4. 

**PART 4**

**1) Is breadth-first generally a better search strategy than depth-first for these inputs? Explain why or why not.**

In general, the time complexity of breadth-first is equal to O(bd), and depth-first is O(bm) (where d=depth of solution, m=max depth of tree). For this type of problem, the goal node is always located at a certain fixed depth (depending on the sentence specification), therefore the depth of solution and max depth of the tree are the same (i.e. d=m). So the time complexity of breadth and depth are both equivalent.

However, the space complexity of depth-first (O(bm)) is better than breadth-first (O(bd)), therefore depth-first is the superior of the two strategies.

**2) Describe the heuristic you chose for the directed search.**

Heuristic function:

Our heuristic function calculates the product of its children’s heuristic value. This is done recursively, until it reaches the depth of the solution (i.e. the goal nodes).

**3) Does your heuristic guarantee that the highest-probability sentence will always be found? Explain why or why not.**

No, because the more nodes you’re multiplying together, the lower the probability will be. So the more descendants (neighbors, neighbors of its neighbors, etc.) that a node has, the lower the heuristic value will be.

**4) Is it possible for your heuristic search to have worse run-time performance than a depth-first or breadth-first search? If worse performance is possible, explain what would cause a worse performance to occur. If not, explain why worse performance would never occur.**

Yes, it is possible for our heuristic to have worse run-time performance than depth and breadth first. For one of the examples (2nd test case from Part 2) that we used heuristic function on, it considered 8258 nodes while depth and breadth both only considered 379 nodes.

One of the reasons for this is that heuristic causes a lot of backtracking, considering more nodes. Another reason is because the heuristic function is recursive, causing the runtime complexity to be larger.