draft.md

### 时间复杂度分析

#### 根据文本生成图

1. **读取和处理文本文件**

public static List<String> processTextFile(String filePath) {  
 StringBuilder text = new StringBuilder();  
  
 try (BufferedReader br = new BufferedReader(new FileReader(filePath))) {  
 String line;  
 while ((line = br.readLine()) != null) {  
 text.append(processLine(line)).append(" ");  
 }  
 } catch (IOException e) {  
 System.out.println("Error reading file: " + e.getMessage());  
 return Collections.emptyList();  
 }  
  
 // 分词  
 String processedText = text.toString().replaceAll("\\s+", " ").trim();  
 return Arrays.asList(processedText.split(" "));  
}

**时间复杂度分析：**

* **读取文件**: 假设文件有 ( L ) 行，每行平均有 ( C ) 个字符，总字符数为 ( N = L \times C )。读取文件的时间复杂度为 ( O(N) )。
* **处理每行字符**: 处理每行字符的时间复杂度为 ( O(C) )，所有行的总时间复杂度为 ( O(N) )。
* **分词**: 分词操作的时间复杂度为 ( O(N) )。

综合起来，读取和处理文本文件的时间复杂度为 ( O(N) )。

1. **构建有向图**

public static Map<String, Map<String, Integer>> buildGraph(List<String> words) {  
 Map<String, Map<String, Integer>> graph = new HashMap<>();  
  
 for (int i = 0; i < words.size() - 1; i++) {  
 String wordA = words.get(i);  
 String wordB = words.get(i + 1);  
  
 graph.putIfAbsent(wordA, new HashMap<>());  
 Map<String, Integer> neighbors = graph.get(wordA);  
 neighbors.put(wordB, neighbors.getOrDefault(wordB, 0) + 1);  
 }  
 graph.putIfAbsent(words.get(words.size() - 1), new HashMap<>());  
 return graph;  
}

**时间复杂度分析：**

* 遍历单词列表：假设单词数为 ( W )，时间复杂度为 ( O(W) )。
* 更新图：每个单词的邻接表更新操作的时间复杂度为均摊 ( O(1) )。

构建图的总时间复杂度为 ( O(W) )。

#### 展示图

public static void printGraph(Map<String, Map<String, Integer>> graph) {  
 for (String wordA : graph.keySet()) {  
 Map<String, Integer> neighbors = graph.get(wordA);  
 for (String wordB : neighbors.keySet()) {  
 System.out.println(wordA + " -> "+wordB + " [weight=" + neighbors.get(wordB) + "]");  
 }  
 }  
}

**时间复杂度分析：**

* 遍历图的每个节点和其邻接节点：假设图中有 ( V ) 个节点和 ( E ) 条边，总时间复杂度为 ( O(V + E) )。

#### 查询桥接词

public static String queryBridgeWords(Map<String, Map<String, Integer>> graph, String word1, String word2) {  
 if (!graph.containsKey(word1) || !graph.containsKey(word2)) {  
 return "No " + (graph.containsKey(word1) ? "word2" : "word1") + " in the graph!";  
 }  
  
 Set<String> bridgeWords = new HashSet<>();  
 Map<String, Integer> neighborsOfWord1 = graph.get(word1);  
  
 for (String potentialBridge : neighborsOfWord1.keySet()) {  
 Map<String, Integer> neighborsOfBridge = graph.get(potentialBridge);  
 if (neighborsOfBridge != null && neighborsOfBridge.containsKey(word2)) {  
 bridgeWords.add(potentialBridge);  
 }  
 }  
  
 if (bridgeWords.isEmpty()) {  
 return "No bridge words from " + word1 + " to " + word2 + "!";  
 } else {  
 return "The bridge words from " + word1 + " to " + word2 + " are: " + String.join(", ", bridgeWords) + ".";  
 }  
}

**时间复杂度分析：**

* 查找和验证词在图中的存在：时间复杂度为 ( O(1) )。
* 遍历第一个词的邻居并查找桥接词：假设第一个词的邻居数为 ( d )，第二个词的邻居数为 ( d' )。总时间复杂度为 ( O(d + d') )。

查询桥接词的总时间复杂度为 ( O(d + d') )。

#### 根据桥接词生成新文本

public static String generateNewText(Map<String, Map<String, Integer>> graph, String newText) {  
 List<String> newWords = Arrays.asList(processLine(newText).split("\\s+"));  
 StringBuilder generatedText = new StringBuilder();  
  
 Random rand = new Random();  
 for (int i = 0; i < newWords.size() - 1; i++) {  
 String word1 = newWords.get(i);  
 String word2 = newWords.get(i + 1);  
 generatedText.append(word1).append(" ");  
  
 Set<String> bridgeWords = new HashSet<>();  
 if (graph.containsKey(word1)) {  
 Map<String, Integer> neighborsOfWord1 = graph.get(word1);  
 for (String potentialBridge : neighborsOfWord1.keySet()) {  
 Map<String, Integer> neighborsOfBridge = graph.get(potentialBridge);  
 if (neighborsOfBridge != null && neighborsOfBridge.containsKey(word2)) {  
 bridgeWords.add(potentialBridge);  
 }  
 }  
 }  
  
 if (!bridgeWords.isEmpty()) {  
 List<String> bridgeWordList = new ArrayList<>(bridgeWords);  
 String bridgeWord = bridgeWordList.get(rand.nextInt(bridgeWordList.size()));  
 generatedText.append(bridgeWord).append(" ");  
 }  
 }  
 generatedText.append(newWords.get(newWords.size() - 1));  
  
 return generatedText.toString();  
}

**时间复杂度分析：**

* 处理输入文本和分词：时间复杂度为 ( O(N) )，其中 ( N ) 为输入文本的字符数。
* 遍历新文本的每对单词：假设新文本有 ( M ) 个单词，遍历每对单词的时间复杂度为 ( O(M) )。
* 查找桥接词：与前述查询桥接词操作相同，时间复杂度为 ( O(d + d') )。

生成新文本的总时间复杂度为 ( O(M \cdot (d + d')) )。

#### 计算最短路径

public static String calcShortestPath(Map<String, Map<String, Integer>> graph, String startWord, String endWord) {  
 if (!graph.containsKey(startWord)) {  
 return "No " + startWord + " in the graph!";  
 }  
  
 Map<String, Integer> distances = new HashMap<>();  
 Map<String, String> predecessors = new HashMap<>();  
 PriorityQueue<String> queue = new PriorityQueue<>(Comparator.comparingInt(distances::get));  
 Set<String> visited = new HashSet<>();  
  
 for (String word : graph.keySet()) {  
 distances.put(word, Integer.MAX\_VALUE);  
 }  
 distances.put(startWord, 0);  
 queue.add(startWord);  
  
 while (!queue.isEmpty()) {  
 String currentWord = queue.poll();  
 if (visited.contains(currentWord)) {  
 continue;  
 }  
 visited.add(currentWord);  
  
 Map<String, Integer> neighbors = graph.get(currentWord);  
 if (neighbors == null) {  
 continue;  
 }  
 int currentDistance = distances.get(currentWord);  
 for (String neighbor : neighbors.keySet()) {  
 int newDist = currentDistance + neighbors.get(neighbor);  
 if (newDist < distances.get(neighbor)) {  
 distances.put(neighbor, newDist);  
 predecessors.put(neighbor, currentWord);  
 queue.add(neighbor);  
 }  
 }  
 }  
  
 if (endWord.isEmpty()) {  
 StringBuilder result = new StringBuilder();  
 for (String word : graph.keySet()) {  
 if (!word.equals(startWord)) {  
 result.append(getPath(startWord, word, predecessors, distances)).append("\n");  
 }  
 }  
 return result.toString();  
 } else {  
 if (!distances.containsKey(endWord) || distances.get(endWord) == Integer.MAX\_VALUE) {  
 return "No path from " + startWord + " to " + endWord + "!";  
 }  
 return getPath(startWord, endWord, predecessors, distances);  
 }  
}

**时间复杂度分析：**

* Dijkstra算法的时间复杂度为 ( O((V + E) \log V) )，其中 ( V ) 是节点数，( E ) 是边数。
* 回溯路径的时间复杂度为 ( O(V) )。

计算最短路径的总时间复杂度为 ( O((V + E) \log V) )。

#### 随机游走

public static String randomWalk(Map<String, Map<String, Integer>> graph) {  
 List<String> nodes = new ArrayList<>(graph.keySet());  
 Random random = new Random();  
 String currentNode = nodes.get(random.nextInt(nodes.size()));  
 Set<String> visitedEdges =  
  
 new HashSet<>();  
 StringBuilder walkPath = new StringBuilder(currentNode);  
  
 while (true) {  
 Map<String, Integer> neighbors = graph.get(currentNode);  
 if (neighbors.isEmpty()) {  
 break;  
 }  
  
 List<String> neighborList = new ArrayList<>(neighbors.keySet());  
 String nextNode = neighborList.get(random.nextInt(neighborList.size()));  
 String edge = currentNode + "->" + nextNode;  
  
 if (visitedEdges.contains(edge)) {  
 break;  
 }  
  
 visitedEdges.add(edge);  
 walkPath.append(" -> ").append(nextNode);  
 currentNode = nextNode;  
 }  
  
 return walkPath.toString();  
}

**时间复杂度分析：**

* 选择起始节点：时间复杂度为 ( O(1) )。
* 随机选择邻居并更新路径：假设随机游走的步数为 ( T )，每步的时间复杂度为 ( O(1) )。

随机游走的总时间复杂度为 ( O(T) )，其中 ( T ) 为随机游走的步数。

### 总结

* **根据文本生成图**: ( O(N) + O(W) )
* **展示图**: ( O(V + E) )
* **查询桥接词**: ( O(d + d') )
* **根据桥接词生成新文本**: ( O(N) + O(M \cdot (d + d')) )
* **计算最短路径**: ( O((V + E) \log V) )
* **随机游走**: ( O(T) )

这些复杂度分析展示了各模块在处理不同大小的输入时的性能预期，帮助我们理解和优化程序。