Aim-Word Ladder II

#include <iostream>

#include <vector>

#include <string>

#include <unordered\_set>

#include <unordered\_map>

#include <queue>

#include <algorithm>

class WordLadderII {

public:

std::vector<std::vector<std::string>> findLadders(std::string beginWord, std::string endWord, std::vector<std::string>& wordList) {

std::unordered\_set<std::string> wordSet(wordList.begin(), wordList.end());

std::vector<std::vector<std::string>> result;

if (wordSet.find(endWord) == wordSet.end()) {

return result; // End word not in wordList

}

std::unordered\_map<std::string, std::vector<std::string>> graph;

std::unordered\_map<std::string, int> distance;

// BFS to build the graph and calculate distances

bfs(beginWord, endWord, wordSet, graph, distance);

// Backtracking to find all shortest paths

std::vector<std::string> path = {beginWord};

backtrack(beginWord, endWord, graph, distance, path, result);

return result;

}

private:

void bfs(const std::string& beginWord, const std::string& endWord, std::unordered\_set<std::string>& wordSet,

std::unordered\_map<std::string, std::vector<std::string>>& graph, std::unordered\_map<std::string, int>& distance) {

std::queue<std::string> q;

q.push(beginWord);

distance[beginWord] = 0;

while (!q.empty()) {

std::string currentWord = q.front();

q.pop();

std::vector<std::string> neighbors = getNeighbors(currentWord, wordSet);

for (const std::string& neighbor : neighbors) {

graph[currentWord].push\_back(neighbor);

if (distance.find(neighbor) == distance.end()) { // First time visiting this word

distance[neighbor] = distance[currentWord] + 1;

q.push(neighbor);

}

}

}

}

std::vector<std::string> getNeighbors(const std::string& word, const std::unordered\_set<std::string>& wordSet) {

std::vector<std::string> neighbors;

std::string tempWord = word;

for (size\_t i = 0; i < word.size(); ++i) {

char originalChar = tempWord[i];

for (char c = 'a'; c <= 'z'; ++c) {

if (c == originalChar) continue;

tempWord[i] = c;

if (wordSet.find(tempWord) != wordSet.end()) {

neighbors.push\_back(tempWord);

}

}

tempWord[i] = originalChar;

}

return neighbors;

}

void backtrack(const std::string& currentWord, const std::string& endWord,

const std::unordered\_map<std::string, std::vector<std::string>>& graph,

const std::unordered\_map<std::string, int>& distance,

std::vector<std::string>& path, std::vector<std::vector<std::string>>& result) {

if (currentWord == endWord) {

result.push\_back(path);

return;

}

if (graph.find(currentWord) == graph.end()) return;

for (const std::string& neighbor : graph.at(currentWord)) {

if (distance.at(neighbor) == distance.at(currentWord) + 1) {

path.push\_back(neighbor);

backtrack(neighbor, endWord, graph, distance, path, result);

path.pop\_back();

}

}

}

};

int main() {

std::string beginWord = "hit";

std::string endWord = "cog";

std::vector<std::string> wordList = {"hot", "dot", "dog", "lot", "log", "cog"};

WordLadderII solver;

std::vector<std::vector<std::string>> result = solver.findLadders(beginWord, endWord, wordList);

std::cout << "All shortest transformation sequences:\n";

for (const auto& sequence : result) {

for (const auto& word : sequence) {

std::cout << word << " ";

}

std::cout << "\n";

}

return 0;

}