Andrew Davison

Dr. William Bailey

CSC 370: Design and Analysis of Algorithms

4/12/2023

Take Home Exam 2

Dynamic Programming – Find the Longest Common Sequence

```
int findLongestCommonSequence(string s1, string s2) {
    int seqArr[s1.length()][s2.length()];

    for (int i = 0; i < s1.length(); i++) {
        for (int j = 0; j < s2.length(); j++) {
            if (s1.chartAt(i) == s2.charAt(j)) {
                if (i == 0 || j == 0) {
                                seqArr[i][j] = 1;
                } else {
                    seqArr[i][j] = seqArr[i-1][j-1];
                }
            } else {
                if (i == 0) {
                    seqArr[i][j] = 0;
                } else {
                    seqArr[i][j] = max(seqArr[i-1][j], seqArr[i][j-1]);
                }
            }
        }
    }

    return seqArr[s1.length() - 1][s2.length() - 1];
}
```

This algorithm solves this problem by looking at how each character in one string compares to the other in an array. The number in the cell is the largest number of matched characters in the substrings that are in the subarray to the upper-left of the entry including the entry itself (selected from the cell directly next to it or directly above it, or 0 if those are not applicable). If the two characters corresponding to the cell in the array from the two strings match, that number is incremented by one. Then, the length of the longest common subsequence is at the very last entry as the numbers build. This works because the matches are only preserved in the cells that proceed father along the strings they are in; otherwise, the algorithm would over count. Instead, each cell represents the largest possible number of matches in a subarray, or largest number of matches possible at the so far observed portions of the strings.

Backtracking – Graph Coloring

```
bool backtrack(int adjacency[][], int partialSol[], int proposedColor) {
    for (int i = 0; i < partialSol.length(); i++) {
        if (adjacency[partialSol.length()][i] > 0 &&
                                partialSol[i] == proposedColor){
                return true;
        }
    }

    return false;
}

bool kColoring(int adjacency[][], int k, int partialSol[]) {
    if (partialSol.length() == adjacency.length()) {
        return true;
    }

    for (int i = 0; i < k; i++) {
        if (!backtrack(adjacency, partialSol, i) {
            partialSol.push_back(i);
            if (kColoring(adjacency, k, partialSol)) {
                return true;
            } else {
```

```
                partialSol.pop_back();
            }
        }
    }

    return false;
}
```

This algorithm works by assigning a color to some node, checking to see if adjacent nodes have been colored that color, and then recursively calling the coloring algorithm on the next node. This algorithm backtracks when it finds an adjacent node to the current node that has the same color as the candidate color for the current node, upon which the algorithm tries a new color for the current node. This continues until either a successful assignment is found or all possible colors cause the algorithm to backtrack, in which case the algorithm goes to the previous node and continues looking for valid color assignments. It is correct to backtrack when a previously colored node is found to be both adjacent to the current node and has the same candidate color, because this partial solution does not satisfy the problem.