perplexity

give me few worst case inputs for me to test that algorithm above in cpp

Below are a few concrete worst-case graph constructions you can use to test your derandomized color-coding algorithm in C++. These examples are chosen because they force the dynamic programming step (which uses subsets of colors) to explore many states—thereby stressing the exponential dependence on k.

Example 1: Complete Graph

A complete graph on n vertices is worst-case because every possible pair of vertices is connected. This leads to a very high edge count (≈ n^2 edges) so that when you run your DP over all edges it must update many table entries. For instance, if you set:

- n = 10
- target path length k = 5 (so you're seeking a colorful path of length 4)

every k-set of vertices appears in many overlapping paths.

Example 2: Layered Directed Acyclic Graph

A layered graph with k layers can force the algorithm to consider an enormous number of candidate paths. In this construction, arrange the vertices into k layers (say, k = 5). Let each layer contain p vertices (for example, p = 4). Then, add an edge from every vertex in layer i to every vertex in layer i + 1. This graph has many combinatorial options:

- The number of simple paths from the first to the last layer is p^(k–1), which greatly increases the number of states your DP must process.

Example 3: Dense "Augmented Chain" Graph

Start with a simple path (a chain) on n vertices (which by itself gives you one long path), and then add many extra "shortcut" edges (for example, an edge from vertex i to vertex i + 2, i + 3, etc.). This creates many alternative routes between nodes on the chain; hence, the DP must consider many overlapping subpaths even though there is an obvious long path.

C++ Code Snippets for Constructing These Graphs

Below are sample code snippets (using an adjacency-list representation) that show how to construct each type of graph.

1. Complete Graph

```
#include &lt;vector&gt;
using namespace std;

vector&lt;vector&lt;int&gt;&gt; createCompleteGraph(int n) {
    // Graph vertices are assumed to be 1-indexed.
    vector&lt;vector&lt;int&gt;&gt; graph(n + 1);
    for (int u = 1; u &lt;= n; u++) {
        for (int v = 1; v &lt;= n; v++) {
            if (u != v) {
                graph[u].push_back(v);
            }
        }
    }
    return graph;
}
```

2. Layered Graph

```
#include &lt;vector&gt;
using namespace std;

vector&lt;vector&lt;int&gt;&gt; createLayeredGraph(int layers, int verticesPerLayer) {
    // Total vertices = layers * verticesPerLayer
    int n = layers * verticesPerLayer;
    vector&lt;vector&lt;int&gt;&gt; graph(n + 1);

    // For each layer, add edges from every vertex to every vertex in the next layer.
    for (int layer = 0; layer &lt; layers - 1; layer++) {
        int startCurrent = layer * verticesPerLayer + 1;
        int startNext    = (layer + 1) * verticesPerLayer + 1;
        for (int i = 0; i &lt; verticesPerLayer; i++) {
            int u = startCurrent + i;
            for (int j = 0; j &lt; verticesPerLayer; j++) {
                int v = startNext + j;
                graph[u].push_back(v);
            }
        }
    }
    return graph;
}
```

3. Augmented Chain Graph

```
#include &lt;vector&gt;
using namespace std;

vector&lt;vector&lt;int&gt;&gt; createAugmentedChain(int n, int extraJump) {
    // Create a chain (path) on n vertices and then add extra edges.
    vector&lt;vector&lt;int&gt;&gt; graph(n + 1);
    for (int i = 1; i &lt; n; i++) {
        // Chain edge
        graph[i].push_back(i + 1);
        // Add extra edges: from vertex i to vertices i+2 up to i+extraJump (if within bo
        for (int jump = 2; jump &lt;= extraJump; jump++) {
            if (i + jump &lt;= n)
                graph[i].push_back(i + jump);
        }
    }
    return graph;
}
```

How to Use These Inputs

1. **Set your target path length k:**
   For example, if you wish to test paths/cycles of length 4 (i.e. 5 vertices), set k = 5 in your algorithm.
2. **Pass the generated graph to your derandomized algorithm:**
   For each worst-case construction, your algorithm will attempt to locate a colorful path of length k–1. Since these graphs have many overlapping candidate paths (complete graph, many layers in the layered graph, or many alternatives in the augmented chain), they serve as stress tests for your dynamic programming routine and the handling of the k-perfect hash family.
3. **Measure runtime and memory usage:**
   Since these graphs are worst-case in terms of path counting and state explosion, they help you gauge the worst-case performance of your implementation.

By testing your algorithm on these graphs (with various sizes for n, layers, and extraJump parameters), you can observe how the algorithm scales in the worst-case scenario and identify any bottlenecks in the DP or hash family evaluation.

⁂