

**now give me a c++ code for this: Develop the derandomized version using precomputed 𝑘-perfect hash families.**

Here's a C++ implementation of the derandomized color-coding algorithm using precomputed k-perfect hash families. This code focuses on finding simple paths of length k-1 in a graph:

#include <iostream>  
#include <vector>  
#include <unordered\_set>  
#include <algorithm>  
#include <cmath>  
  
using namespace std;  
  
class KPerfectHashFamily {  
 int n; // Maximum vertex ID  
 int k; // Target subgraph size  
 int prime; // Prime number > n  
 vector<pair<int, int>> hash\_params; // (a, b) pairs for hash functions  
  
public:  
 KPerfectHashFamily(int max\_vertex, int target\_size) : n(max\_vertex), k(target\_size) {  
 prime = next\_prime(n);  
   
 // Generate hash parameters (simplified construction)  
 for(int a = 1; a <= k; a++) {  
 for(int b = 0; b < k; b++) {  
 hash\_params.emplace\_back(a, b);  
 }  
 }  
 }  
  
 // Get color assignment for current hash function  
 vector<int> get\_colors(int func\_index) const {  
 auto [a, b] = hash\_params[func\_index];  
 vector<int> colors(n+1); // vertices are 1-based  
   
 for(int v = 1; v <= n; v++) {  
 colors[v] = ((a \* v + b) % prime) % k + 1;  
 }  
 return colors;  
 }  
  
 size\_t num\_functions() const { return hash\_params.size(); }  
  
private:  
 int next\_prime(int start) {  
 while(true) {  
 start++;  
 if(is\_prime(start)) return start;  
 }  
 }  
  
 bool is\_prime(int num) {  
 if(num <= 1) return false;  
 for(int i = 2; i <= sqrt(num); i++)  
 if(num % i == 0) return false;  
 return true;  
 }  
};  
  
class DerandomizedColorCoding {  
 const vector<vector<int>> adj; // adjacency list  
 const int k; // path length to find (k-1 edges)  
 KPerfectHashFamily hash\_family;  
  
public:  
 DerandomizedColorCoding(const vector<vector<int>>& graph, int target\_size)  
 : adj(graph), k(target\_size), hash\_family(graph.size()-1, target\_size) {}  
  
 bool contains\_path() {  
 // Try all hash functions in the family  
 for(size\_t fi = 0; fi < hash\_family.num\_functions(); fi++) {  
 auto colors = hash\_family.get\_colors(fi);  
 if(has\_colorful\_path(colors)) return true;  
 }  
 return false;  
 }  
  
private:  
 bool has\_colorful\_path(const vector<int>& colors) {  
 // DP table: dp[v][mask] = exists path to v with color mask  
 vector<vector<bool>> dp(adj.size(), vector<bool>(1 << k, false));  
   
 // Initialize for single-node paths  
 for(int v = 1; v < adj.size(); v++) {  
 int mask = 1 << (colors[v] - 1);  
 dp[v][mask] = true;  
 }  
  
 // Dynamic programming for paths of increasing length  
 for(int len = 1; len < k; len++) {  
 for(int v = 1; v < adj.size(); v++) {  
 for(int mask = 0; mask < (1 << k); mask++) {  
 if(!dp[v][mask] || \_\_builtin\_popcount(mask) != len) continue;  
   
 for(int u : adj[v]) {  
 int color\_bit = 1 << (colors[u] - 1);  
 if(!(mask & color\_bit)) {  
 dp[u][mask | color\_bit] = true;  
 }  
 }  
 }  
 }  
 }  
  
 // Check if any vertex has a full color mask  
 for(int v = 1; v < adj.size(); v++) {  
 if(dp[v][(1 << k) - 1]) return true;  
 }  
 return false;  
 }  
};  
  
int main() {  
 // Example graph (1-based vertices)  
 vector<vector<int>> graph = {  
 {}, // 0 unused  
 {2, 3}, // 1  
 {1, 3, 4}, // 2  
 {1, 2, 4}, // 3  
 {2, 3, 5}, // 4  
 {4} // 5  
 };  
  
 int target\_length = 4; // Look for path with 4 edges (5 vertices)  
   
 DerandomizedColorCoding dcc(graph, target\_length);  
 cout << "Contains path of length " << target\_length-1 << ": "  
 << boolalpha << dcc.contains\_path() << endl;  
  
 return 0;  
}

Key components:

1. **KPerfectHashFamily**:
   * Generates a family of hash functions using simple linear parameters
   * Uses prime number modulus to create color distributions
   * Provides deterministic color assignments for vertices
2. **DerandomizedColorCoding**:
   * Iterates through all hash functions in the family
   * For each coloring, uses dynamic programming to detect colorful paths
   * Maintains DP state tracking color usage masks
3. **Colorful Path Detection**:
   * Uses bitmasking to track color usage
   * Dynamic programming approach similar to randomized version
   * Checks for full color coverage (all k colors used)

The hash family construction in this implementation is simplified for demonstration. A production implementation would:

* Use a more sophisticated hash family construction (e.g., from[[1]](#fn1))
* Handle larger values of k through better parameter generation
* Include optimizations for sparse graphs and memory usage

This code shows the fundamental approach of derandomizing color-coding through precomputed hash families while maintaining the core algorithmic structure of the original randomized method.

⁂

1. <https://web.math.princeton.edu/~nalon/PDFS/balancedf.pdf>