**Time/Space complexity analysis**

|  |  |  |
| --- | --- | --- |
| **Function** | **Time/Comment** | **Space/Comment** |
| int main() | O(mn2 + mn)   * O(mn2)   Where mn2 represents the time complexity of getPairableCompanies() and mn represents the complexity of reading user input. m represents number of subcompanies, n represents num of components | O(n2)  Where n2 represents the space complexity of the constructor function, which allocates a 2d vector to store the graph |
| bool Backtrack::isPairable(vector, int, int) | O(n)  This function will at worst iterate through all components of the graph, represented by n | O(1)  No new space is being allocated |
| bool Backtrack::backtrackLogic(vector, int) | O(m \* n2)   * O(mn2)   Where n represents the number of components in the graph, and m represents the number of colors. This function will at worst iterate through each component n times, to check for m amount of assignable colors. | O(n)  Where n represents the number of components. It will at worst loop through all components to check if it can be assigned a color, and since it’s a recursive function, it’ll add function calls to the stack in the process |
| Backtrack::Backtrack(int, int, int) | O(1)  No computations are being performed | O(n2)  Where n represents the number of components. This constructor will allocate a 2d array. |
| void Backtrack::getPairableCompanies() | O(mn2)  This complexity represents the complexity of the backtrackLogic() function, which is the function with the worst time complexity inside this function. | O(n)  Where n represents the number of components. This function utilizes this complexity at 2 places: once when it initializes the colors vector, and once when it calls the backtrackLogic() function. |
| void Backtrack::addEdge(int, int) | O(1)  Only one operation is performed | O(1)  No new space is allocated |

**Test Plan**

|  |  |  |  |
| --- | --- | --- | --- |
| **Reason for test** | **Actual input data** | **Expected output data** | **Actual output** |
| Check if no solution logic works. If all components connect to each other, then there can be no solution since the number of colors needed to assign to each component > number of subcompanies. | 3 2  2 3 0  1 3 0  1 2 0 | no solution | no solution |
| Check if no solution logic works. This test will check for components that are all not connected to each other. This specific test case will need at most 2 colors, but 1 subcompany, so no solution | 4 1  4 0  3 0  2 0  1 0 | no solution | No solution |
| Check for basic performance capabilities. It took about half a second to load | 30 2  2 16 0  3 1 0  4 2 0  5 3 0  6 4 0  7 5 0  8 6 0  9 7 0  10 8 0  11 9 0  12 10 0  13 11 0  14 12 0  15 13 0  30 14 0  1 17 0  16 18 0  17 19 0  18 20 0  19 21 0  20 22 0  21 23 0  22 24 0  23 25 0  24 26 0  25 27 0  26 28 0  27 29 0  28 30 0  15 29 0 | 1 3 5 7 9 11 13 15 17 19 21 23 25 27 29  2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 | 1 3 5 7 9 11 13 15 17 19 21 23 25 27 29  2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 |
| Check for basic program functionality to see if it can correctly color the graph and group lexicographically | 8 4  2 6 5 4 0  1 6 7 3 0  2 7 4 8 0  1 3 8 5 0  8 4 1 6 0  5 1 2 7 0  6 2 3 8 0  7 4 5 3 0 | 1 3  2 4  5 7  6 8 | 1 3  2 4  5 7  6 8 |
| Check for basic program functionality to see if it can correctly color the graph and group lexigoraphically | 10 3  5 6 2 0  1 7 3 0  2 8 4 0  5 9 3 0  4 10 1 0  1 9 8 0  2 10 9 0  6 10 3 0  4 6 7 0  5 7 8 0 | 1 3 7  2 4 6 10  5 8 9 | 1 3 7  2 4 6 10  5 8 9 |