

Graph Isomorphism

The problem of graph isomorphism is one that has been discussed for many years. Many people have worked on various aspects of it, but as for a bound on the time complexity of an algorithm to determine whether two graphs are isomorphic, the most recent breakthrough is due to the mathematician László Babai, who showed that the problem can be done in "quasipolynomial" time ($\exp(\ln^{O(1)} n)$) in [this paper](#).

I demonstrate my own much much much less elegant (but straightforward) algorithm that runs in worse than factorial time.

Setup

I define a Graph as an object which encapsulates a `std::vector<std::vector<bool>>`, representing the adjacency matrix of the graph.

I use booleans assuming that the maximum number of edges between any pair of vertices is one.

This Graph object is constructed from a string which represents the rows of the adjacency matrix, separated by commas.

I overload `operator<<` to display the matrix, and then `operator==` to check whether two graphs are equal, asserting that any isomorphic graphs are also equal.

Implementation

Formally, two graphs G and H are isomorphic if

$$\exists f: V(G) \rightarrow V(H), \forall uv \in E(G) \Leftrightarrow f(u)f(v) \in E(H)$$

Which is kind of opaque to think about.

Computationally, it's more obvious that this describes the existence of a "vertex permutation" from G to H .

essentially, if we label the vertices of G from 1 to n (assuming they both have n vertices), we need to find a way to permute this labelling such that matrix of G becomes the matrix of H .

Upon realizing this we can use an implementation of [Heap's Algorithm](#) (NB: this only generates permutations of an array, it's not related to graphs directly in any way) to generate the permutations and check them each manually.

It's useful to realize at this point that parts of our matrix representation are redundant, and we can ignore the main diagonal because we are assuming that no "loops" are present, and we only need to check one side of this diagonal, because we assume an un-directed graph, so we have symmetry along this diagonal.

Program Output

```
main.cpp (~Desktop\Folder_university\Tings\Discrete\DMprogHW) - VIM
69 if (foundPermutation) {
70     for (int i = 0; i < lhs.size; ++i)
71         cout << i << " " << perm[i] << " ";
72     cout << "\n";
73 }
74 return foundPermutation;
75 }
76
77 int main() {
78     // Simple example:
79     /*
80      0 - 1
81      |
82      2   3
83      */
84     Graph graph1 {"0100,1001,0000,0100"};
85     /*
86      0 1
87      | /
88      2 3
89      */
90     Graph graph2 {"0010,0010,1100,0000"};
91     cout << graph1 << graph2 << (graph1 == graph2 ? "True" : "False") << "\n\n";
92
93     // Popular pentagon-pentagram example:
94     graph1 = {"01001,10100,01010,00101,10010"}; // Pentagon
95     graph2 = {"00110,00011,10001,11000,01100"}; // Pentagram
96     cout << graph1 << graph2 << (graph1 == graph2 ? "True" : "False") << "\n\n";
97
98     // A pair of Visually "similar" but non-isomorphic graphs:
99     /*
100      0 ----- 3
101      |         |
102      1 - 2     |
103      |         |
104      4 ----- 7
105      |         |
106      5 - 6     |
107      |         |
108      4 ----- 7
109      */
110     graph1 = {"01011000,10100100,01000010,10000001,10000001,01000010,00100101,00011010"};
111     /*
112      0 ----- 3
113      |         |
114      1 - 2     |
115      |         |
116      4 ----- 7
117      |         |
118      5 - 6     |
119      |         |
120      4 ----- 7
121      */
122     graph2 = {"01011000,10100100,01000010,10000001,10000101,01001010,00100100,00011000"};
123     cout << graph1 << graph2 << (graph1 == graph2 ? "True" : "False") << "\n";
124 }
125
126 PS C:\Users\Emmanuel K\Desktop\Folder_university\Tings\Discrete\DMprogHW> g++ (ls *.cpp) -O3 -o main.exe
127 PS C:\Users\Emmanuel K\Desktop\Folder_university\Tings\Discrete\DMprogHW> .\main
128 0100
129 1001
130 0000
131 0100
132
133 0010
134 0010
135 1100
136 0000
137
138 0-0, 1-2, 2-3, 3-1,
139 True
140
141 01001
142 10100
143 01010
144 00101
145 10010
146
147 00110
148 00011
149 10001
150 11000
151 01100
152
153 0-0, 1-2, 2-3, 3-2, 4-3,
154 True
155
156 01011000
157 10100100
158 01000010
159 10000001
160 10000001
161 10000001
162 01000010
163 00100101
164 00011010
165
166 01011000
167 10100100
168 01000010
169 10000001
170 10000001
171 10000101
172 01001010
173 00100100
174 00011000
175
176 False
177 PS C:\Users\Emmanuel K\Desktop\Folder_university\Tings\Discrete\DMprogHW>
```