Ullman's Algorithm for Subgraph isomorphism match

The subgraph isomorphism problem asks whether a graph G has a subgraph $G' \subset G$ that is isomorphmic to a graph P. So basically you have the picture on the box of a $puzzle(G)$ and want to know where a particular $piece(P)$ fits, if at all. It is **NP-complete** because Hamiltonian cycle is a special case.

In 1976 Ullman proposed a backtracking algorithm for this problem. The writeup in the original paper is hard to follow because the pseudo-code doesn't use functions or even loops. Maybe those weren't invented back then.

The basic algorithm

It is possible to encode a subgraph isomorphism as a $|V_P| \times |V_G|$ matrix M in which each row contains exactly one 1 and each column contains at most one 1. We set m_{ij} to 1 iff $v_j \in G$ corresponds to $v_i \in P$ in the isomorphism. Then $P = M(MG)^T$, where I use P and G to stand for the adjacency matrices. If we aren't looking for induced subgraphs, $P \leq M(MG)^T$ (componentwise), i.e. the subgraph of G selected by M might contain additional edges not present in P.

The algorithm works by systematically enumerating possible matrices M and checking whether they actually encode an isomorphism.

We start by setting up a $|V_P| \times |V_G|$ matrix M^0 that contains a 1 at (i,j) if it is possible that $v_i \sim v_j$ in some subgraph isomorphism. For now, we only use the degree as a criterion, i.e. we can map v_i to v_j if the latter has enough neighbors: $m^0_{ij} = 1 \Leftrightarrow \deg(v_i) \leq \deg(v_j)$. If we were cleverer we could remove more 1's and reduce our runtime.

Now all we need to do is try all matrices M that can be obtained from M^0 by removing all but one 1 from each row while having at most one 1 in each column. We do that recursively.

```
Recurse (used_columns, cur_row, G, P, M)
        if cur_row = num_rows(M)
                if M is an isomorphism:
                        output yes and end the algorithm
        M' = M
        prune(M')
        for all unused columns c
                set column c in M' to 1 and other columns to 0
                mark c as used
                recurse(used_column, cur_row+1, G, P, M')
                mark c as unused
        output no
```

Making the M' copy of M isn't really necessary, but it will become so once we implement our pruning procedure.

Pruning

We want to (safely) change at least some of the 1's in our matrix to 0's to reduce the computation time. For that, we use a simple observation. If some $p \in V_P$ has neighbours $p_1, \ldots, p_l \in P$, and we map it to some $g \in V_G$, then we'd better also map $p_1, \ldots, p_l$ to neigbors of g.

Remember that a 1 at (i, j) in M means that we still think that $v_i \in P$ can correspond to $v_j \in G$. But if we already found out that there is a neighbour of $v_i \in P$ can't be mapped to any neigbour of $v_j \in G$ clearly the 1 at (i, j) is wrong and we can change it safely to a 0. This change might make more mappings impossible, so we iterate this check until nothing can be changed. If we remove all 1's from a row during this refinement, we can stop the whole process, since M can't be completed to an isomorphism anymore.

```
do
        for all (i,j) where M is 1
                for all neighbors x of vi in P
                        if there is no neighbor y of vj s.t. M(x,y)=1
                                M(i,j)=0
while M was changed
```

Now the effectiveness of our pruning procedure depends on the order of the vertices. The earlier in the recursion we find a 1 that can be changed to a 0, the better. Thus it is a good a idea to order the vertices such that high degree vertices of P are first.

Clever Implementation This algorithm seems rather costly, since we do a matrix multplication for every leaf in the recursion tree and manipulate the matrix quite a lot in between. However, note that we're dealing with boolean matrices here. It's a good idea to encode them as bit vectors. Then multiplication can be done efficiently using bit-twiddling, even if we still use the naive $O(n^3)$ algorithm. Similarly setting a column to 1 and the other columns to 0 can be done using bit-twiddling and finding viable neighbors during the pruning step is fast too. Using these implementation tricks we can speed up the naive algorithm by some largish constant factor, depending on the word size of the CPU and whether it supports SSE or similar vector operations.