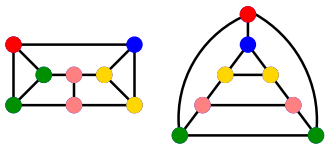
Module 7 project:
Graph Isomorphism

Part II: Branching Algorithms



Ruben Hoeksma, slides partly by Paul Bonsma

What happens now? Group part of the project

Project groups:

- Everyone has been assigned a group.
- If you have not contacted your group yet, do this as soon as possible.

Deliverables:

- Team contract (this week).
- Buddycheck surveys (midway: week 6; final: week 10).
- Competition and submission of code (17 April).
- Submission of report (18 April).
- Keep track of performance of (different versions of) your algorithm for the report/documentation.

What happens now? Group part of the project

You have implemented Colour Refinement.

You will extend those algorithms to solve Graph Isomorphism and Counting Automorphisms, as a group.

Three more **lectures**:

- Branching (today).
- Fast Colour Refinement.
- Automorphism Groups.

Six **project sessions** to get help with implementation/questions.

Summary last lecture

Recall the *Graph Isomorphism (GI) Problem*, and the *colour refinement* algorithm to *efficiently* compute a *coarsest stable colouring* of a given graph.

Definition

A colouring α of G is stable if for all $u, v \in V(G)$ it holds that:
if $\alpha(u) = \alpha(v)$, then u and v have identically coloured neighborhoods.

Key fact:

Theorem

For every graph G and partition π_0 of $V(G)$, there is a unique coarsest stable partition of $V(G)$ that refines π_0, which is the partition given by colour refinement, when starting with π_0.

Colour Refinement Invariant

A useful invariant that follows from last lecture:

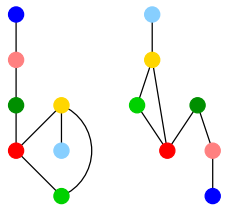
Proposition (Invariant)

Let α be a colouring of a graph $G \uplus H$, and let β be the stable colouring of $G \uplus H$ given by colour refinement, when starting with α.
Then any isomorphism $f : V(G) \to V(H)$ that is colour preserving for α is also colour preserving for β.

Possible choices for the "start colouring" α are *uniform colouring* or colouring by *vertex degree*, but today we will also see alternatives.

Colour Refinement Outcome: Example

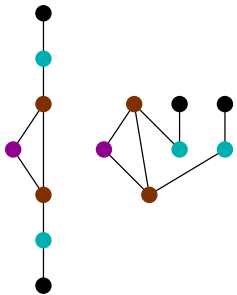
Let α be the stable colouring of $G \uplus H$ that results from applying colour refinement, starting with the *uniform colouring*.



Observation

If α defines a bijection $f : V(G) \to V(H)$, then f is the unique *isomorphism from G to H.*

Better than Enumeration: Recursive Colour Refinement



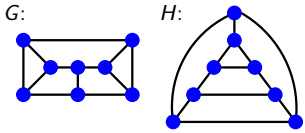
Better idea: Choose a vertex x of G for which $f(x)$ is not yet clear.
"Fix" which vertex y of H it should be mapped to. Give x and y a new colour, and apply colour refinement, again.
(Every guess corresponds to a branch of a *recursive algorithm*.)

In the example: only two guesses necessary.

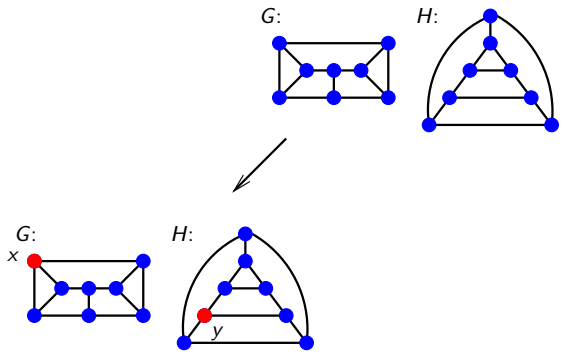
Individualization Refinement - Overview

- α: balanced stable colouring of $G \uplus H$, does not define a bijection.
- Exists colour class C with $2k$ vertices, $k \geq 2$ (k vertices of G and k vertices of H). Denote $C_G = C \cap V(G)$ and $C_H = C \cap V(H)$.
- Choose $x \in C_G$. Every colour preserving isomorphism f has $f(x) = y$ for some $y \in C_H$.
- Try out all such possibilities for $f(x)$ (gives k *branches* of a recursive algorithm): *There exists a colour preserving isomorphism if and only if a colour preserving isomorphism will be found in at least one branch.*
- In each branch (*kmany*), the *choice* $f(x) = y$ is encoded by giving both vertices a *new, unique colour*.
- Given this start colouring, we can apply colour refinement again, and continue recursively until either an isomorphism is found, or it is concluded that no isomorphism with $f(x) = y$ exists.

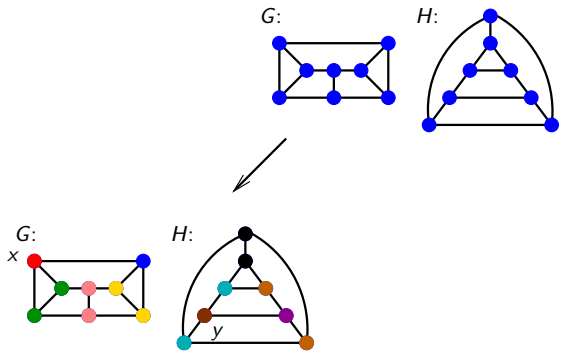
Example



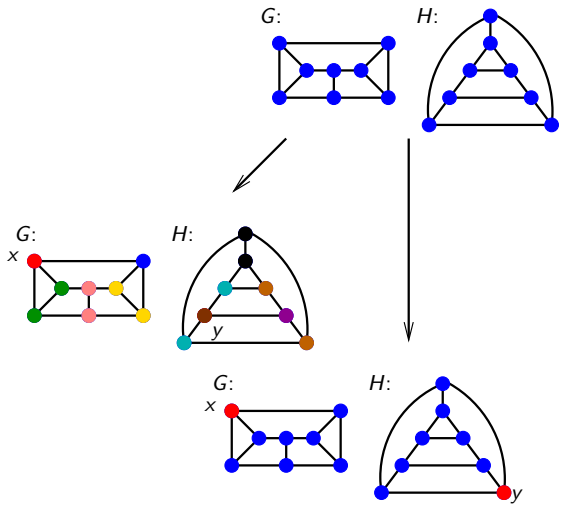
Example



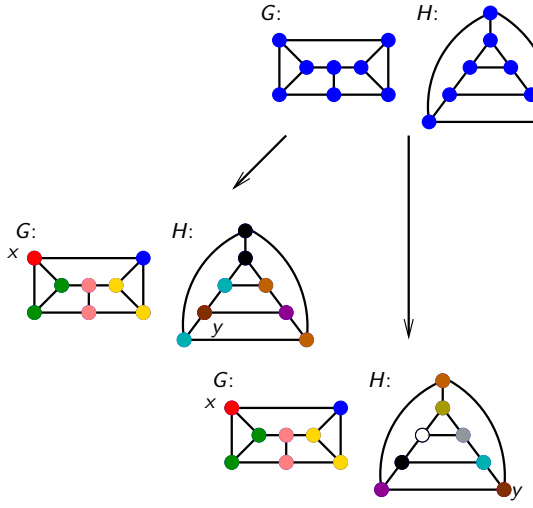
Example



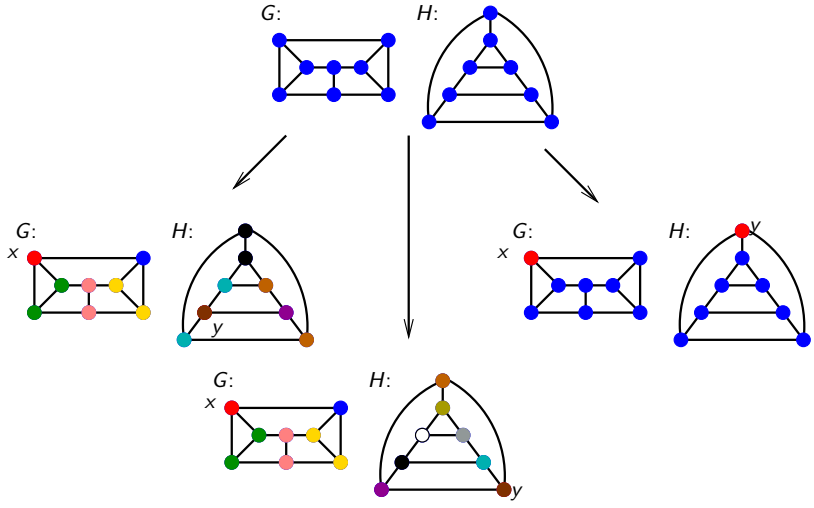
Example



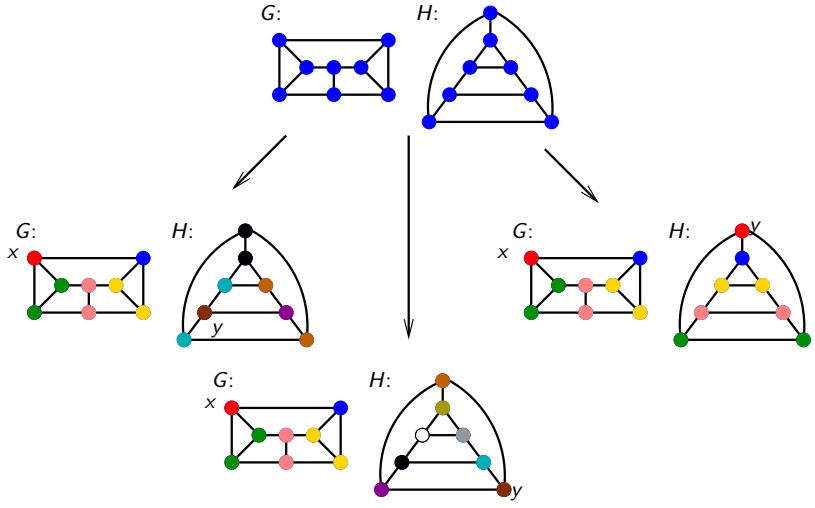
Example



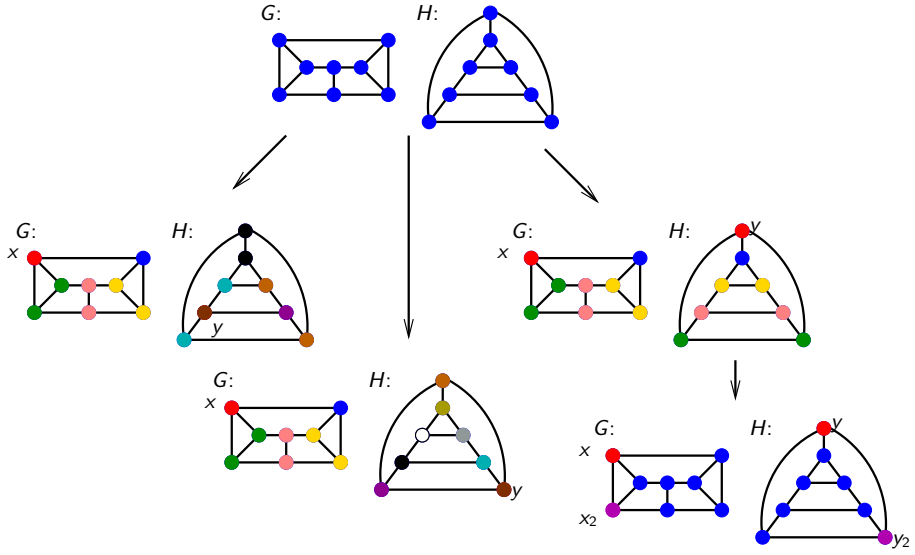
Example



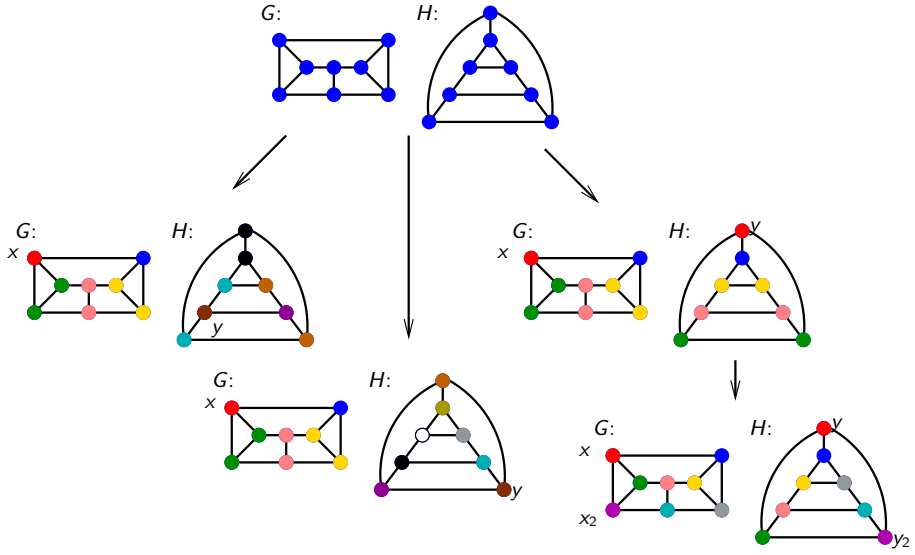
Example



Example



Example



Individualization Refinement - Implementation Details

- Let $D = (x_1, \ldots, x_d)$ and $I = (y_1, \ldots, y_d)$ be sequences of distinct vertices of G and H, respectively.
- A bijection $f : V(G) \to V(H)$ *follows* D, I if for all $i \in \{1, \ldots, d\}$: $f(x_i) = y_i$.
- Let $\alpha(D, I)$ be the colouring of $G \uplus H$ that assigns colour i to x_i and y_i for $i \in \{1, \ldots, d\}$, and colour 0 to all vertices not in $D \cup I$.

Proposition

Let β be the coarsest stable colouring of $G \uplus H$ that refines $\alpha(D, I)$.
Every isomorphism f from G to H that follows D, I is β-color preserving.

Corollary

If β unbalanced, $\nexists$ isomorphism from G to H following D, I. If β defines bijection f, f is the unique *isomorphism from G to H following D, I.*

Algorithm 2: Individualization Refinement

Subroutine CountIsomorphism(D, I):
Input: D and I are equal length sequences of vertices of G resp. H.
Output: The number of isomorphisms from G to H that follow D, I.

Compute the coarsest stable colouring β of $G \uplus H$ that refines $\alpha(D, I)$
if β is unbalanced:
    return 0
if β defines a bijection:
    return 1
Choose a colour class C with $|C| \geq 4$.
Choose $x \in C \cap V(G)$.
num $= 0$
for all $y \in C \cap V(H)$:
    num := num+CountIsomorphism$(D + x, I + y)$
return num

Correctness of Individualization Refinement

Theorem

Algorithm 2 is correct; it returns the number of isomorphisms from G to H that follow D, I.

Proof Sketch: By induction over the recursion tree:
Base: the unbalanced / bijection case (use the corollary).
Induction step: use the previous proposition. □

Theorem

Algorithm 2 terminates.

Proof Sketch: In every recursive call D and I grow by one element.
If $|D| = |I| = |V(G)| = |V(H)|$, the algorithm is done. Thus, the recursion depth is at most $|V(G)|$ and the algorithm terminates. □

Summary of Individualization Refinement

- Algorithm 2 computes the number of isomorphisms between two graphs G and H. (When called with D and I both empty.)
- The number of recursive calls is at least the number of isomorphisms between G and H, which can be exponential. (worst case: $n!$)
- Implementation for GI (decision problem): terminate as soon as one isomorphism is found!
- By choosing $H := G$, we can solve the #Aut problem (counting the number of automorphisms of G).

Improvement 1: Branching Rules

Improvements are possible by choosing the colour class C and vertex x for branching cleverly.

Implementation idea: Try out and compare different *branching rules*.
Find at least one rule that performs better than straightforward rules, and support this with computational experiments.

Improvement 2: Preprocessing

Examples of problematic structures:

Definition

Two vertices $u, v \in V(G)$ are twins if $uv \in E(G)$ and $N(u)\backslash\{v\} = N(v)\backslash\{u\}$. They are false twins if $N(u) = N(v)$.

Observation

Let α be a colouring that assigns the same colour to two (false) twins u, v. The coarsest stable colouring that refines α also assigns same colour to u and v.

If G has a set of k (false) twins, then this can add a factor $k!$ to the number of recursive calls made by Algorithm 2!
Implementation Idea: use *preprocessing algorithm*[1] that detects (false) twins before applying individualization refinement.

[1] You have to come up with the algorithm yourself.

Improvement 3: Trees or Forests

If G and H are trees or forests, our branching algorithm works very well for the GI problem, but not so well for the #Aut problem, when there are many automorphisms.

Implementation Idea: It can be shown that branching algorithm solves the GI problem for trees in *polynomial time*. However there is also a polynomial time algorithm[2] for the #Aut problem on trees...

[2] You can search the internet for more information on what this algorithm is.

Project: This Week

- Implement in Python an individualization refinement algorithm, that can correctly solve the GI and the #Aut Problem.
- This requires the *colour refinement* algorithm to work with any initial colouring - if you have not implemented this, you need to start there.
- On canvas, there are additional *instances* of the GI and #Aut problems for testing, which require individualization refinement.
- Use small instances (for verifying correctness), and large instances (for tuning the performance).
- Think about which additional tricks you want to implement (twins, trees/forests, etc.). Test how they influence performance.

Milestone:

- End of this week, Python program that can correctly solve the GI problem and #Aut problem for "small" instances.
- Along the way: *improving the computational complexity* is useful (for getting *bonus points* at the end). Line profiler plugin for PyCharm can be useful.