Computing Isomorphism

[Ch.7, Kreher & Stinson] [Ch.3, Kaski & Östergård]

Lucia Moura

Winter 2009

Isomorphism of Combinatorial Objects

In general, *isomorphism* is an equivalence relation on a set of objects.
When generating combinatorial objects, we are often interested in **generating inequivalent objects**:

> **Generate exactly one representative of each isomorphism class.**

(We don't want to have isomorphic objects in our list.)

For example, when interested in graphs with certain properties, the labels on the vertices may be irrelevant, and we are really interested on the unlabeled underlying structure.

Isomorphism can be seen as a general equivalence relation, but for combinatorial objects, isomorphism is defined through the existence of an appropriate bijection (isomorphism) that shows that two objects have the same structure.

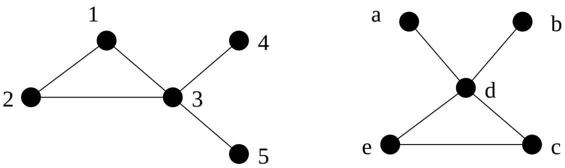
What are the issues in Isomorphism Computations?

- Isomorphism: decide whether two objects are isomorphic.
  Some approaches:
  - Compute an **isomorphism invariant** for an object
    If two objects disagree on the invariant, then the objects are NOT isomorphic; the converse is not true.
  - Compute a **certificate** for an object
    Two objects are isomorphic if and only if they agree on the certificate.
  - Put an object on **canonical form**
    Two objects are isomorphic if and only if they have the same canonical form.
- Automorphism group generators: compute generators of the automorphism group of an object.

We can go a long way with coloured graphs

- We will concentrate on graphs and coloured graphs (= a graph plus a partition of the vertex set).
- Most combinatorial objects can be represented as coloured graphs.
- We then reduce the isomorphism of more general combinatorial objects to the isomorphism of coloured graphs.
- Brendan McKay's *nauty* software (short for "no automorphism, yes?", available online) is an extremely efficient package/C procedure for isomorphism of graphs and coloured graphs. It is based on backtracking and partition refinement ideas and uses the same framework we will study here to compute certificates for graphs.
- In the next lecture notes chapter "Isomorph-free exhaustive generation", we will use isomorphism computations studied in this chapter as black boxes.

Example 1: isomorphic graphs



G_1 and G_2 are **isomorphic**, since there is a bijection $f : V_1 \to V_2$ that preserve edges:

$$\begin{array}{ccc} 1 & \to & c \\ 2 & \to & e \\ 3 & \to & d \\ 4 & \to & a \\ 5 & \to & b \end{array}$$

Example 2: non-isomorphic graphs

G_3 and G_4 are not isomorphic:
Any bijection would not preserve edges since G_3 has no vertex of degree 3, while G_4 does.
(the degree sequence of a graph (in sorted order) is invariant under isomorphism)

Definition of graph isomorphism and automorphism

Definition

Two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are **isomorphic** if there is a bijection $f : V_1 \rightarrow V_2$ such that

$$\{f(x), f(y)\} \in E_2 \iff \{x, y\} \in E_1.$$

The mapping f is said to be an **isomorphism** between G_1 and G_2.
If f is an isomorphism from G to itself, it is called an **automorphism**.

The set of all automorphisms of a graph is a permutation group (which is a group under the "composition of permutations" operation). See chapter 6 for more on groups and permutation groups.

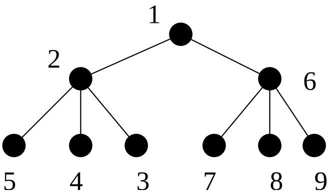
Computational complexity of graph isomorphism

The problem of determining if two graphs are isomorphic is in general difficult, but most researchers believe it is not NP-complete.

Some special cases can be solved in polynomial time, such as: graphs with maximum degree bounded by a constant and trees.

An example of invariant

Let $DS = [deg(v_1), deg(v_2), \ldots, deg(v_n)]$ be the degree sequence of a graph; let $SDS = [d_1, d_2, \ldots, d_n]$ be its degree sequence in sorted order.



SDS is the same for all graphs that are isomorphic to G.

So, SDS is an *invariant* (under isomorphism).

Definition of Invariant

Definition

Let $\mathcal{F}$ be a family of graphs. An *invariant* on $\mathcal{F}$ is a function ϕ with domain $\mathcal{F}$ such that $\phi(G_1) = \phi(G_2)$ if G_1 is isomorphic to G_2.

- If $\phi(G_1) \neq \phi(G_2)$ we can conclude G_1 and G_2 are not isomorphic. If $\phi(G_1) = \phi(G_2)$, we still need to check whether they are isomorphic.

Definition of Invariant

Definition

Let $\mathcal{F}$ be a family of graphs. An *invariant* on $\mathcal{F}$ is a function ϕ with domain $\mathcal{F}$ such that $\phi(G_1) = \phi(G_2)$ if G_1 is isomorphic to G_2.

- If $\phi(G_1) \neq \phi(G_2)$ we can conclude G_1 and G_2 are not isomorphic. If $\phi(G_1) = \phi(G_2)$, we still need to check whether they are isomorphic.
- Invariants can help us to quickly determine when two structures are not isomorphic, and so avoiding a full isomorphism test.

Definition of Invariant

Definition

Let $\mathcal{F}$ be a family of graphs. An *invariant* on $\mathcal{F}$ is a function ϕ with domain $\mathcal{F}$ such that $\phi(G_1) = \phi(G_2)$ if G_1 is isomorphic to G_2.

- If $\phi(G_1) \neq \phi(G_2)$ we can conclude G_1 and G_2 are not isomorphic. If $\phi(G_1) = \phi(G_2)$, we still need to check whether they are isomorphic.
- Invariants can help us to quickly determine when two structures are not isomorphic, and so avoiding a full isomorphism test.
- Examples of invariants: number of vertices and edges, degree sequence, number of components, etc.

Definition of Invariant

Definition

Let $\mathcal{F}$ be a family of graphs. An *invariant* on $\mathcal{F}$ is a function ϕ with domain $\mathcal{F}$ such that $\phi(G_1) = \phi(G_2)$ if G_1 is isomorphic to G_2.

- If $\phi(G_1) \neq \phi(G_2)$ we can conclude G_1 and G_2 are not isomorphic. If $\phi(G_1) = \phi(G_2)$, we still need to check whether they are isomorphic.
- Invariants can help us to quickly determine when two structures are not isomorphic, and so avoiding a full isomorphism test.
- Examples of invariants: number of vertices and edges, degree sequence, number of components, etc.
- To be useful, invariants should be quickly computable. "Number of cliques" is an invariant, but not quickly computable.

Invariant inducing function

Definition (vertex partition induced by a function)

Let $\mathcal{F}$ be a family of graphs on the vertex set V.
Let $D : \mathcal{F} \times V \to \{0, 1, \ldots, k\}$.
Then, the **partition of V induced by D** is

$$B = [B[0], B[1], \ldots, B[k]]$$

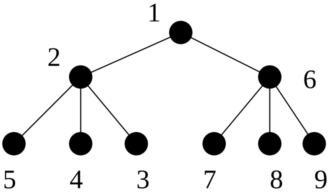
where $B[i] = \{v \in V : D(G, v) = i\}$.

Definition (invariant inducing function)

If $\phi_D(G) = [|B[0]|, |B[1]|, \ldots, |B[k]|]$ is an invariant (under isomorphism), then we say that D is an **invariant inducing** function.

Example: invariant inducing function

$D(G,u) =$ degree of vertex u in graph G.



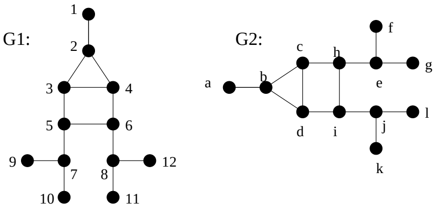
Ordered partition induced by D:

$$B = [\emptyset, \{3,4,5,7,8,9\}, \{1\}, \emptyset, \{2,6\}, \emptyset, \emptyset, \emptyset, \emptyset]$$

$$\phi_D(G) = [0,6,1,0,2,0,0,0,0]$$

$\phi_D(G)$ is an invariant for $\mathcal{F}$, the family of all graphs on V.
So, D is an invariant inducing function.

Using more than one invariant inducing function



$D_1(G, v)$ = tuple representing the # of neighbours for each degree
Ex.: $D_1(G_1, 4) = [0030 \cdots 0]$; $D_1(G_2, b) = [0030 \cdots 0]$;
$D_1(G_1, 8) = [2010 \cdots 0]$

$D_2(G, v)$ = # of triangles in G passing through v.
Ex.: $D_2(G_1, 4) = 1$; $D_1(G_2, b) = 0$.

Partition refinement using two invariant inducing functions

Compute an (ordered) vertex partition where the corresponding tuple of sizes is an invariant under isomorphism.
If two graphs disagree on the tuple of sizes, then they are not isomorphic.
Otherwise, we can use the ordered partition to reduce the number of permutations considered.

- Initial partition: $X_0(G_1) = [\{1, 2, \ldots, 12\}]$ $X_0(G_2) = [\{a, b, \ldots, l\}]$
- Partition refinement of X_0 induced by D_1:
  $X_1(G_1) = [\{1, 9, 10, 11, 12\}, \{2\}, \{3, 4, 5, 6\}, \{7, 8\}]$
  $X_1(G_2) = [\{a, f, g, k, l\}, \{b\}, \{c, d, h, i\}, \{e, f\}]$
- Partition refinement of X_1 induced by D_2:
  $X_2(G_1) = [\{1, 9, 10, 11, 12\}, \{2\}, \{3, 4\}, \{5, 6\}, \{7, 8\}]$
  $X_2(G_2) = [\{a, f, g, k, l\}, \{b\}, \{c, d\}, \{h, i\}, \{e, j\}]$
- G_1 and G_2 are still compatible; but we only need to check bijections that map vertices from $X_2(G_1)[i]$ into vertices of $X_2(G_2)[i]$, $1 \leq i \leq 5$.

$D_1(G,v) = \#$ of neighbours for each degree

$[0010\cdots 0] = D_1(G_1,1) = D_1(G_1,9) = D_1(G_1,10) = D_1(G_1,11) = D_1(G_1,12)$
$[1020\cdots 0] = D_1(G_1,2)$
$[0030\cdots 0] = D_1(G_1,3) = D_1(G_1,4) = D_1(G_1,5) = D_1(G_1,6)$
$[2010\cdots 0] = D_1(G_1,7) = D_1(G_1,8)$

$[0010\cdots 0] = D_1(G_2,a) = D_1(G_2,f) = D_1(G_2,g) = D_1(G_2,k) = D_1(G_2,l)$
$[1020\cdots 0] = D_1(G_2,b)$
$[0030\cdots 0] = D_1(G_2,c) = D_1(G_2,d) = D_1(G_2,h) = D_1(G_2,i)$
$[2010\cdots 0] = D_1(G_2,e) = D_1(G_2,f)$

Partition refinement of X_0 induced by D_1:

$$X_1(G_1) = [\{1,9,10,11,12\},\{2\},\{3,4,5,6\},\{7,8\}]$$
$$X_1(G_2) = [\{a,f,g,k,l\},\{b\},\{c,d,h,i\},\{e,f\}]$$

$X_1(G_1) = [\{1,9,10,11,12\},\{2\},\{3,4,5,6\},\{7,8\}]$
$X_1(G_2) = [\{a,f,g,k,l\},\{b\},\{c,d,h,i\},\{e,f\}]$

$D_2(G,v) = \#$ of triangles in G passing through v.

$$\begin{aligned} D_2(G_1,v) &= 0, \quad \text{if } v \in \{1,5,6,7,8,9,10,11,12\} \\ &= \mathbf{1}, \quad \text{if } v \in \{2,3,4\} \\ D_2(G_2,v) &= 0, \quad \text{if } v \in \{a,e,f,g,h,i,j,k,l\} \\ &= \mathbf{1}, \quad \text{if } v \in \{b,c,d\} \end{aligned}$$

Partition refinement of X_1 induced by D_2:

$$\begin{aligned} X_2(G_1) &= [\{1,9,10,11,12\},\{2\},\{\mathbf{3},\mathbf{4}\},\{5,6\},\{7,8\}] \\ X_2(G_2) &= [\{a,f,g,k,l\},\{b\},\{\mathbf{c},\mathbf{d}\},\{h,i\},\{e,j\}] \end{aligned}$$

G_1 and G_2 are still compatible!

Looking at the partition of the vertex set obtained by the two invariant induction functions:

$X_2(G_1) = [\{1, 9, 10, 11, 12\}, \{2\}, \{\mathbf{3}, \mathbf{4}\}, \{5, 6\}, \{7, 8\}]$
$X_2(G_2) = [\{a, f, g, k, l\}, \{b\}, \{\mathbf{c}, \mathbf{d}\}, \{h, i\}, \{e, f\}]$

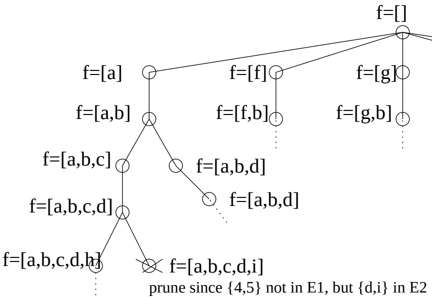
We only need to check bijections between sets in corresponding cells (colours):

$$\begin{array}{rcl} \{1, 9, 10, 11, 12\} & \leftrightarrow & \{a, f, g, k, l\} \\ \{2\} & \leftrightarrow & \{b\} \\ \{\mathbf{3}, \mathbf{4}\} & \leftrightarrow & \{\mathbf{c}, \mathbf{d}\} \\ \{5, 6\} & \leftrightarrow & \{h, i\} \\ \{7, 8\} & \leftrightarrow & \{e, f\} \end{array}$$

# of bijections to test: $5! \times 1! \times 2! \times 2! \times 2! = 960$.
Without partition refinement, we would have to test $12!$ bijections!

Backtracking algorithm to find all isomorphisms

We use a set $\mathcal{I}$ of invariant inducing functions, and then apply backtracking in order to generate all valid bijections.



Algorithm $\text{ISO}(\mathcal{I}, G_1, G_2)$ (global n, W, X, Y)
procedure $\text{GETPARTITIONS}()$

$X[0] \leftarrow V(G_1)$; $Y[0] \leftarrow V(G_2)$; $N \leftarrow 1$;
for each $D \in \mathcal{I}$ do
 for $i \leftarrow 0$ to $N-1$ do
  Partition $X[i]$ into sets $X_1[i], X_2[i], \ldots, X_{m_i}[i]$,
   where $x, x' \in X_j[i] \iff D(x) = D(x')$
  Partition $Y[i]$ into sets $Y_1[i], Y_2[i], \ldots, Y_{n_i}[i]$,
   where $y, y' \in Y_j[i] \iff D(y) = D(y')$
  if $m_i \neq n_i$ then exit; (G_1 and G_2 are not isomorphic)
  Order $Y_1[i], Y_2[i], \ldots, Y_{m_i}[i]$ so that for all j
   $D(x) = D(y)$ whenever $x \in X_j[i]$ and $y \in Y_j[i]$
  if ordering is not possible then exit; (not isomorphic)
 Order the partitions so that:
  $|X[i]| = |Y[i]| \leq |X[i+1]| = |Y[i+1]|$ for all i
 $N \leftarrow N + (m_0 - 1) + \ldots + (m_{N-1} - 1)$;
return (N);

```
procedure FindIsomorphism(l)
    if l = n then output (f);
    j ← W[l];
    for each y ∈ Y[j] do
        OK ← true;
        for u ← 0 to l − 1 do
            if ({u, l} ∈ E(G_1) and {f[u], y} ∉ E(G_2)) or
              ({u, l} ∉ E(G_1) and {f[u], y} ∈ E(G_2)) then OK ← false;
        if OK then f[l] ← y;
                        FindIsomorphism(l + 1);
main
  N ←GetPartitions();
  for i ← 0 to N do for each x ∈ X[i] do W[x] ← i;
  FindIsomorphism(0);
```

Certificates

Definition

A *certificate* $Cert()$ for a family $\mathcal{F}$ of graphs is a function such that for $G_1, G_2 \in \mathcal{F}$, we have

$$Cert(G_1) = Cert(G_2) \iff G_1 \text{ and } G_2 \text{ are isomorphic}$$

Next, we show how to compute certificates in polynomial time for the family of **trees**.

Consequently, graph isomorphism for trees can be solved in polynomial time!

Certificates for Trees

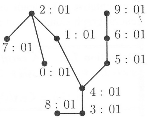
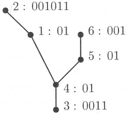
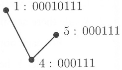
Algorithm to compute certificates for a tree:

1. Label all vertices with string 01.
2. While there are more than 2 vertices in G:
   for each non-leaft x of G do
   1. Let Y be the set of labels of the leaves adjacent to x and the label of x with initial 0 and trailing 1 deleted from x;
   2. Replace the label of x with the concatenation of the labels in Y, sorted in increasing lexicographic order, with a 0 prepended and a 1 appended.
   3. Remove all leaves adjacent to x.
3. If there is only one vertex x left, report x's label as the certificate.
4. If there are 2 vertices x and y left, concatenate x and y in increasing lexicographic order, and report it as the certificate.

Example 1:

tree to certificate

Example 7.2 *A tree with one center*

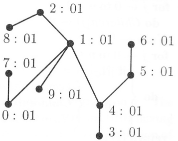
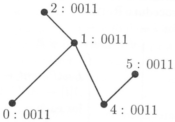
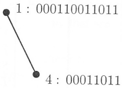
Number of Vertices	Non-leaves	Current Tree
9	1 : $Y = \{\}$ 2 : $Y = \{01, 01\}$ 3 : $Y = \{01\}$ 4 : $Y = \{\}$ 5 : $Y = \{\}$ 6 : $Y = \{01\}$	
6	1 : $Y = \{001011\}$ 4 : $Y = \{0011\}$ 5 : $Y = \{0011\}$	
3	4 : $Y = \{00010111, 000111, 0011\}$	
1		• 4 : 00001011100011100111

Certificate = 00001011100011100111. ▯

Example 2:

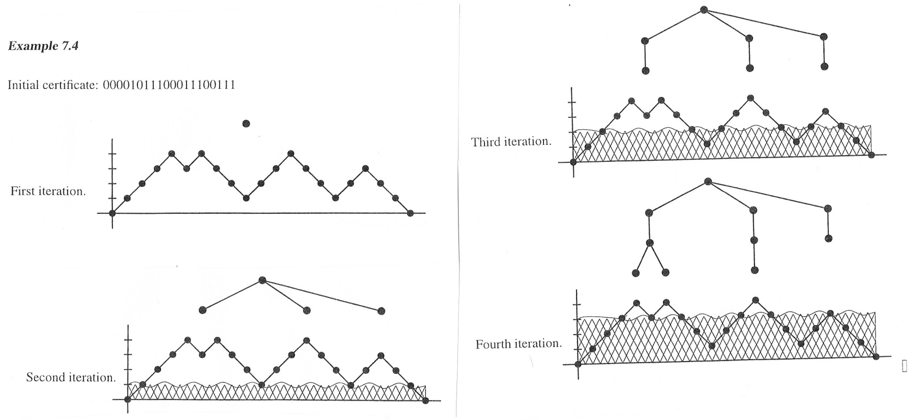
tree to certificate

Example 7.3 *A tree with two centers*

Number of Vertices	Non-leaves	Current Tree
9	0 : $Y = \{01\}$ 1 : $Y = \{01\}$ 2 : $Y = \{01\}$ 4 : $Y = \{01\}$ 5 : $Y = \{01\}$	
5	1 : $Y = \{0011, 0011, 01\}$ 4 : $Y = \{0011, 01\}$	
2		

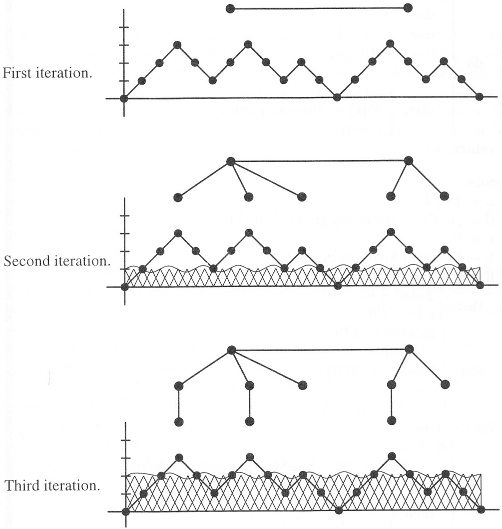
Certificate = 00011001101100011011.

Example 1: certificate to tree



Example 2:

certificate to tree



Certificates for general graphs

Let $G = (V, E)$. Consider all permutations $\pi : V \to V$.
Each π determines an adjacency matrix:

$$A_\pi[u, v] = \begin{cases} 1, & \text{if } \{\pi(u), \pi(v)\} \in E \\ 0, & \text{otherwise.} \end{cases}$$

Look at the relevant entires of A_π and form a number Num_π.
We will use these Num_π to define a certificate...

Example: adjacency matrices for isomorphic graphs

$$G = (V = \{1,2,3\}, E = \{\{1,2\},\{1,3\}\})$$

π :	A_π :	Num_π	π :	A_π :	Num_π
$[1,2,3]$	$\begin{bmatrix} - & 1 & 1 \\ - & - & 0 \\ - & - & - \end{bmatrix}$	110	$[1,3,2]$	$\begin{bmatrix} - & 1 & 1 \\ - & - & 0 \\ - & - & - \end{bmatrix}$	110
$[2,1,3]$	$\begin{bmatrix} - & 1 & 0 \\ - & - & 1 \\ - & - & - \end{bmatrix}$	101	$[2,3,1]$	$\begin{bmatrix} - & 0 & 1 \\ - & - & 1 \\ - & - & - \end{bmatrix}$	011
$[3,1,2]$	$\begin{bmatrix} - & 1 & 0 \\ - & - & 1 \\ - & - & - \end{bmatrix}$	101	$[3,2,1]$	$\begin{bmatrix} - & 0 & 1 \\ - & - & 1 \\ - & - & - \end{bmatrix}$	011

Defining a certificate for general graphs: idea 1

- We could define the certificate to be

$$Cert1(G) = min\{Num_{\pi}(G) : \pi \in Sym(V)\}.$$

Defining a certificate for general graphs: idea 1

- We could define the certificate to be

$$Cert1(G) = min\{Num_{\pi}(G) : \pi \in Sym(V)\}.$$

- $Cert1(G)$ is difficult to compute.

Defining a certificate for general graphs: idea 1

- We could define the certificate to be

$$Cert1(G) = min\{Num_\pi(G) : \pi \in Sym(V)\}.$$

- $Cert1(G)$ is difficult to compute.
- $Cert1(G)$ has as many leading 0's as possible.

Defining a certificate for general graphs: idea 1

- We could define the certificate to be

$$Cert1(G) = min\{Num_\pi(G) : \pi \in Sym(V)\}.$$

- $Cert1(G)$ is difficult to compute.
- $Cert1(G)$ has as many leading 0's as possible.
- So, k is as large as possible, where k is the number of all-zero columns above the diagonal.

Defining a certificate for general graphs: idea 1

- We could define the certificate to be

$$Cert1(G) = min\{Num_{\pi}(G) : \pi \in Sym(V)\}.$$

- $Cert1(G)$ is difficult to compute.
- $Cert1(G)$ has as many leading 0's as possible.
- So, k is as large as possible, where k is the number of all-zero columns above the diagonal.
- So, vertices $\{1, 2, \ldots, k\}$ form a maximum independent set in G (or equivalently a maximum clique in the complement graph $\overline{G}$).

Defining a certificate for general graphs: idea 1

- We could define the certificate to be

$$Cert1(G) = min\{Num_\pi(G) : \pi \in Sym(V)\}.$$

- $Cert1(G)$ is difficult to compute.
- $Cert1(G)$ has as many leading 0's as possible.
- So, k is as large as possible, where k is the number of all-zero columns above the diagonal.
- So, vertices $\{1, 2, \ldots, k\}$ form a maximum independent set in G (or equivalently a maximum clique in the complement graph $\overline{G}$).
- So, computing $Cert1(G)$ as defined above is NP-hard.

Defining a certificate for general graphs: idea 1

- We could define the certificate to be

$$Cert1(G) = min\{Num_\pi(G) : \pi \in Sym(V)\}.$$

- $Cert1(G)$ is difficult to compute.
- $Cert1(G)$ has as many leading 0's as possible.
- So, k is as large as possible, where k is the number of all-zero columns above the diagonal.
- So, vertices $\{1, 2, \ldots, k\}$ form a maximum independent set in G (or equivalently a maximum clique in the complement graph $\overline{G}$).
- So, computing $Cert1(G)$ as defined above is NP-hard.
- But it is believed that determining if $G_1 \sim G_2$ (G_1 isomorphic to G_2) is not NP-complete.

Defining a certificate for general graphs: idea 1

- We could define the certificate to be

$$Cert1(G) = min\{Num_\pi(G) : \pi \in Sym(V)\}.$$

- $Cert1(G)$ is difficult to compute.
- $Cert1(G)$ has as many leading 0's as possible.
- So, k is as large as possible, where k is the number of all-zero columns above the diagonal.
- So, vertices $\{1, 2, \ldots, k\}$ form a maximum independent set in G (or equivalently a maximum clique in the complement graph $\overline{G}$).
- So, computing $Cert1(G)$ as defined above is NP-hard.
- But it is believed that determining if $G_1 \sim G_2$ (G_1 isomorphic to G_2) is not NP-complete.
- So, it is possible that the approach of computing $Cert1(G)$ to solve the graph isomorphism problem is more work than necessary.

Defining a certificate for general graphs

So, instead, we will define the certificate as follows:

$$Cert(G) = min\{Num_{\pi}(G) : \pi \in \Pi_G\},$$

where Π_G is a set of permutations determined by the structure of G but not by any particular ordering of V.

This is what we do next.

The main idea is to do partition refinement, and use backtracking whenever we reach an equitable partition (partition that can't be further refined). The minimum above is taken over permutations considered in this backtracking tree.

Discrete and equitable partitions

Definition

A partition B is a **discrete partition** if $|B[j]| = 1$ for all j, $0 \leq j \leq k$.
It is a **unit partition** if $|B| = 1$.

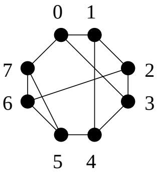
Definition

Let $G = (V, E)$ be a graph and $N_G(u) = \{x \in V : \{u, x\} \in E\}$.
A partition B is an **equitable partition** with respect to the graph G if for all i and j

$$|N_G(u) \cap B[j]| = |N_G(v) \cap B[j]|$$

for all $u, v \in B[i]$.

Given B an ordered equitable partition with k blocks, we can define M_B to be a $k \times k$ matrix where $M_B[i,j] = |N(G(v)) \cap B[j]|$ where $v \in B[i]$.
(Since B is equitable any choice of v produces the same result)
Define $Num(B) :=$ sequence of $k(k-1)/2$ elements above diagonal written column by column.
$B = [\{0\},\{2,4\},\{5,6\},\{7\},\{1,3\}]$ is an equitable partition w.r.to G:



$$M_B = \begin{bmatrix} 0 & 0 & 0 & 1 & 2 \\ 0 & 0 & 1 & 0 & 2 \\ 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 2 & 0 & 0 \\ 1 & 2 & 0 & 0 & 0 \end{bmatrix}$$

and $Num(B) = [0,0,1,1,0,1,2,2,0,0]$.

If B is a **discrete** partition then B corresponds to a permutation $\pi : B[i] = \{\pi[i]\}$, in which case $Num(B) = Num_\pi(G)$, adjusting so that $Num(B)$ is interpreted as the sequence of bits of a binary number.

Partition Refinement

Definition

An ordered partition B is a **refinement** of the ordered partition A if

1. every block $B[i]$ of B is contained in some block $A[j]$ of A; and
2. if $u \in A[i_1]$ and $v \in A[j_1]$ with $i_1 \leq j_1$, then $u \in B[i_2]$ and $v \in B[j_2]$ with $i_2 \leq j_2$.

The definition basically says that B must refine A and preserve its order.
$A = [\{0,3\},\{1,2,4,5,6\}]$
$B = [\{0,3\},\{1,5,6\},\{2,4\}]$ is a refinement of A,
$B' = [\{1,5,6\},\{2,4\},\{0,3\}]$ is not a refinement of A (blocks out of order)
Let A be an ordered partition and T be any block of A.
Define $D_T : V \to \{0,1,\ldots,n-1\}$, $D_T(v) = |N_G(v) \cap T|$.
This function can be used to refine A.

Computing and equitable partition

1. Set B equal to A.
2. Let $\mathcal{S}$ be a list containing the blocks of B.
3. While $(\mathcal{S} \neq \emptyset)$ do
4. remove a block T from the list $\mathcal{S}$
5. for each block $B[i]$ of B do
6. for each h, set $L[h] = \{v \in B[i] : D_T(v) = h\}$
7. if there is more than one non-empty block in L then
8. replace $B[i]$ with the non-empty blocks in L in order of the index h, $h = 0, 1, \ldots, n-1$.
9. add the non-empty blocks in L to the end of the list $\mathcal{S}$

Notes:
In step 4 we ignore blocks of $\mathcal{S}$ if the block has already been partitioned in B.
The procedure will produce an equitable partition.
The ordering at step 8 is chosen in order to make $Num(B)$ smaller.

Algorithm for partition refinement

Algorithm 7.5 $\textsc{Refine}(n, \mathcal{G}, A, B)$ (global L, U, S, T, N)

procedure $\textsc{SplitAndUpdate}(n, \mathcal{G}, B, j)$

$L \leftarrow$ empty list

for each $u \in B[j]$ do $\{ h \leftarrow |T \cap N_{\mathcal{G}}(u)|;\ L[h] \leftarrow L[h] \cup \{u\};\ \}$

$m \leftarrow 0$

for $h \leftarrow 0$ to $n-1$ do if $L[h] \neq \emptyset$ then $m \leftarrow m+1$

if $m > 1$ then

for $h \leftarrow |B|-1$ downto $j+1$ do $B[m-1+h] \leftarrow B[h]$

$k \leftarrow 0$

for $h \leftarrow 0$ to $n-1$ do

if $L[h] \neq \emptyset$ then $B[j+k] \leftarrow L[h];\ S[N+k] \leftarrow L[h];$
$U \leftarrow U \cup L[h];\ k \leftarrow k+1;$

$j \leftarrow j+m-1$

$N \leftarrow N+m$

Algorithm for partition refinement (cont'd)

main
$B \leftarrow A$
for $N \leftarrow 0$ to $|B|$ do $S[N] \leftarrow B[N]$
$U \leftarrow \mathcal{V}$
while $N \neq 0$ do
  $N \leftarrow N - 1$
  $T = S[N]$
  if $T \subset U$ then
    $U \leftarrow U \setminus T$
    $j \leftarrow 0$
    while $j < |B|$ and $|B| < n$ do
      if $|B| \neq 1$ then $\textsc{SplitAndUpdate}(n, \mathcal{G}, B, j)$
      $j \leftarrow j + 1$
    if $|B| = n$ then exit

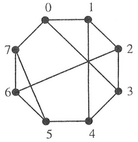
Example 7.7

Example 7.7 *Refining to an equitable partition*
We illustrate the refinement procedure using the graph given in 7.6 and the initial partition $A = [\{0\}, \{1,2,3,4,5,6,7\}]$.

$$B = [\{0\}, \{1,2,3,4,5,6,7\}]$$
$$S = [\{1,2,3,4,5,6,7\}, \underbrace{\{0\}}_{T}]$$
$$D_{\{0\}} : B = [\{0\}, \{2,4,5,6\}, \{1,3,7\}]$$
$$S = [\{1,2,3,4,5,6,7\}, \{1,3,7\}, \underbrace{\{2,4,5,6\}}_{T}]$$
$$D_{\{2,4,5,6\}} : B = [\{0\}, \{2,4\}, \{5,6\}, \{1,3,7\}]$$
$$S = [\{1,2,3,4,5,6,7\}, \{1,3,7\}, \{5,6\}, \underbrace{\{2,4\}}_{T}]$$
$$D_{\{2,4\}} : B = [\{0\}, \{2,4\}, \{5,6\}, \{7\}, \{1,3\}]$$
$$S = [\{1,2,3,4,5,6,7\}, \{1,3,7\}, \{5,6\}, \{1,3\}, \underbrace{\{7\}}_{T}]$$
$$D_{\{7\}} : B = [\{0\}, \{2,4\}, \{5,6\}, \{7\}, \{1,3\}]$$
$$S = [\{1,2,3,4,5,6,7\}, \{1,3,7\}, \{5,6\}, \underbrace{\{1,3\}}_{T}]$$
$$D_{\{1,3\}} : B = [\{0\}, \{2,4\}, \{5,6\}, \{7\}, \{1,3\}]$$
$$S = [\{1,2,3,4,5,6,7\}, \{1,3,7\}, \underbrace{\{5,6\}}_{T}]$$
$$D_{\{5,6\}} : B = [\{0\}, \{2,4\}, \{5,6\}, \{7\}, \{1,3\}]$$
$$S = [\{1,2,3,4,5,6,7\}, \underbrace{\{1,3,7\}}_{T}]$$
$$S = [\underbrace{\{1,2,3,4,5,6,7\}}_{T}]$$
$$S = [\quad]$$

The final refined equitable partition is $B = [\{0\}, \{2,4\}, \{5,6\}, \{7\}, \{1,3\}]$. □

Example 7.6 *An equitable partition*



$$B = [\{0\}, \{2,4\}, \{5,6\}, \{7\}, \{1,3\}]$$

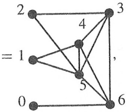
is an equitable partition,

$$M_B = \begin{bmatrix} 0&0&0&1&2 \\ 0&0&1&0&2 \\ 0&1&1&1&0 \\ 1&0&2&0&0 \\ 1&2&0&0&0 \end{bmatrix}, \text{ and}$$

$$\text{Num}(B) = [0,0,1,1,0,1,2,2,0,0].$$

Example 7.8: incomplete - needs to explore possibles discrete partitions that refine equitable...

Example 7.8 *Refining to discrete partitions*
We illustrate the refinement procedure using the graph

$\mathcal{G} =$ 

and the initial partition $A = [\{0,1,2,3,4,5,6\}]$. The adjacency matrix of $\mathcal{G}$ is

0	0	0	0	0	0	1
0	0	0	0	1	1	0
0	0	0	1	0	1	0
0	0	1	0	1	1	1
0	1	0	1	0	1	1
0	1	1	1	1	0	1
1	0	0	1	1	1	0

and

$$\mathrm{Num}_{\mathbf{I}}(\mathcal{G}) = (000001010101111100111)_{\text{binary}}.$$

$$B = [\{0,1,2,3,4,5,6\}]$$

$$D_{\{0,1,2,3,4,5,6\}} : B = [\{0\},\{1,2\},\{3,4,6\},\{5\}]$$
$$S = [\{5\},\{3,4,6\},\{1,2\},\underbrace{\{0\}}_{T}]$$

$$D_{\{0\}} : B = [\{0\},\{1,2\},\{3,4\},\{6\},\{5\}]$$
$$S = [\{5\},\{3,4,6\},\{1,2\},\{6\},\underbrace{\{3,4\}}_{T}]$$

$$D_{\{3,4\}} : B = [\{0\},\{1,2\},\{3,4\},\{6\},\{5\}]$$
$$S = [\{5\},\{3,4,6\},\{1,2\},\underbrace{\{6\}}_{T}]$$

$$D_{\{6\}} : B = [\{0\},\{1,2\},\{3,4\},\{6\},\{5\}]$$
$$S = [\{5\},\{3,4,6\},\underbrace{\{1,2\}}_{T}]$$

$$D_{\{1,2\}} : B = [\{0\},\{1,2\},\{3,4\},\{6\},\{5\}]$$
$$S = [\{5\},\underbrace{\{3,4,6\}}_{T}]$$

$$D_{\{3,4,6\}} : B = [\{0\},\{1,2\},\{3,4\},\{6\},\{5\}]$$
$$S = [\underbrace{\{5\}}_{T}]$$

$$D_{\{5\}} : B = [\{0\},\{1,2\},\{3,4\},\{6\},\{5\}]$$

Defining a certificate: backtracking + partition refinement

We will examine **Algorithm 7.8:** $\textsc{Cert1}(\mathcal{G})$, which will calculate:

$$Cert(G) = min\{Num_\pi(G) : \pi \in \Pi_G\},$$

where Π_G is a set of permutations determined by the structure of G but not by any particular ordering of V.

The main idea is to use backtracking combined with partition refinement.

At each node, we do partition refinement until we reach an equitable partition; at this point, if the partition is not discrete, we branch on elements of the first block whose size is greater than one.

Each element of this block gives rise to a branch where this element will be chosen to be first in the discrete partition.

The minimum above is taken over permutations considered in the backtracking tree that we have just defined (not over all possible permutations); these permutations are determined by the graph structure and not by any particular ordering of V.

Algorithm for computing a certificate for general graphs

Algorithm 7.8: $\textsc{Cert1}(\mathcal{G})$ external $\textsc{Canon1}()$

$P \leftarrow [\{0, 1, \ldots, n\}]$

$\textsc{Canon1}(\mathcal{G}, P)$ (Main algorithm: get $Best$ for canonical adjacency matrix

(Next steps: Convert matrix for $Best$ into number (certificate) C:

$C := Num_{Best}(\mathcal{G}) = \min\{Num_\pi(\mathcal{G}) : \pi \in \Pi(G)\}$)

$k \leftarrow 0$; $C \leftarrow 0$

for $j \leftarrow n - 1$ downto 1 do

  for $i \leftarrow j - 1$ downto 0 do

    if $\{Best[i], Best[j]\} \in \mathcal{E}(\mathcal{G})$ then $C \leftarrow x + 2^k$

    $k \leftarrow k + 1$

return (C)

Algorithm 7.7: $\textsc{Canon1}(\mathcal{G}, P)$ external $\textsc{Refine}()$, $\textsc{Compare}()$

$\textsc{Refine}(n, \mathcal{G}, P, Q)$
Find the index l of the first block of Q with $|Q[l]| > 1$
$Res \leftarrow Better$
if $BestExist$ then for $i \leftarrow 0$ to $l-1$ do $\pi_1[i] \leftarrow q_i$, where $Q[i] = \{q_i\}$
$Res \leftarrow \textsc{Compare}(\mathcal{G}, \pi_1, l)$
if Q has n blocks then
if **not** $BestExist$ then
$BestExist \leftarrow \mathbf{true}$; for $i \leftarrow 0$ to $n-1$ do $Best[i] \leftarrow q_i$, where $Q[i] = \{q_i\}$
else if $Res = Better$ then $Best \leftarrow \pi_1$
else if $Res \neq Worse$ then
$ChoicesLeft \leftarrow Q[l]$; $AllChoices \leftarrow Q[l]$ (branch on refinement of $Q[l]$)
for $j \leftarrow 0$ to $l-1$ do $R[j] \leftarrow Q[j]$
for $j \leftarrow l+1$ to $size(Q)$ do $R[j+1] \leftarrow Q[j]$
while $ChoicesLeft \neq \emptyset$ do
$u \leftarrow$ any element of $ChoicesLeft$
$R[l] \leftarrow \{u\}$; $R[l+1] \leftarrow AllChoices \setminus \{u\}$
$\textsc{Canon1}(\mathcal{G}, R)$
$ChoicesLeft \leftarrow ChoicesLeft \setminus \{u\}$

This algorithm compares the first l numbers of permutations π and $Best$, to decide whether π may lead to lexicographical smaller number than $Best$.

Algorithm 7.6: $\textsc{Compare}(\mathcal{G}, \pi, l)$

for $j \leftarrow 1$ to $l-1$ do
  for $i \leftarrow 0$ to $j-1$ do
    $x \leftarrow A_{\mathcal{G}}[Best[i], Best[j]]$
    $y \leftarrow A_{\mathcal{G}}[\pi[i], \pi[j]]$
    if $x < y$ then return $(Worse)$
    if $x > y$ then return $(Better)$
return $(Equal)$

Backtracking for a certificate for general graphs

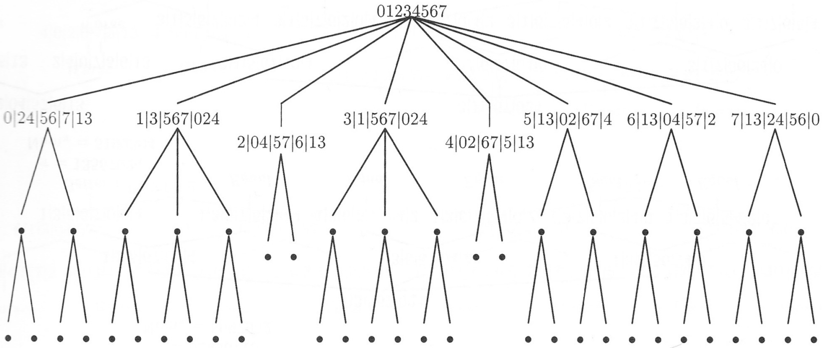


FIGURE 7.2
Overview of the state space tree that results from running Algorithm 7.7 on the graph in Example 7.7.

Backtracking for a certificate for general graphs

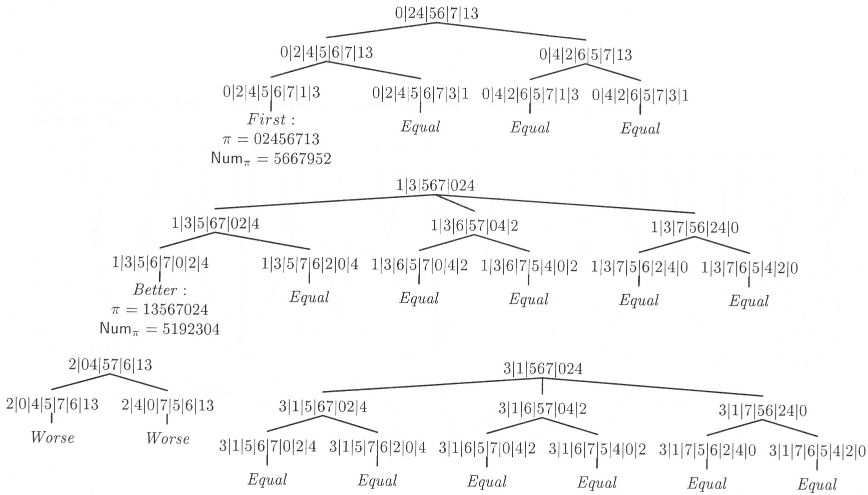
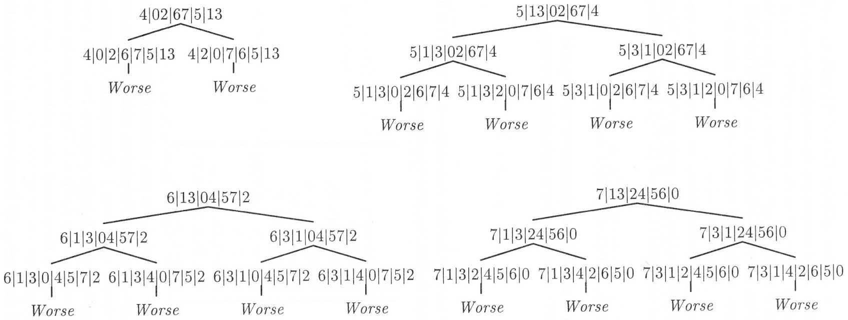


FIGURE 7.2 (continued)
Subtrees with roots 0|24|56|7|13, 1|3|567|024, 2|04|57|13 **and** 3|1|567|024.



$|G| = 12$, Certificate $=$ 5192304, and NODES $= 55$.

FIGURE 7.2 (continued)
Subtrees with roots 4|02|67|5|13, 5|13|02|67|4, 6|13|04|57|2 **and** 7|13|24|56|0.

Pruning with Automorphisms

Let $G = (V, E)$ and $\pi \in Sym(V)$, a permutation on V.
Recall that π is an automorphism of G if it is an isomorphism from G to itself.
Let A be the adjacency matrix of G and let A_π the the adjacency matrix of G with respect to a permutation π, that is, $A_\pi[i,j] = A[\pi[i], \pi[j]]$, for all i, j. Then, π is an automorphism of G if and only if $A_\pi = A$.

Theorem

If $Num_{\pi_1}(G) = Num_\mu(G)$ then $\pi_2 = \pi_1\mu^{-1}$ is an automorphism of G.

PROOF.

$$\begin{aligned} A_{\pi_2}[i,j] &= A_{\pi_1\mu^{-1}}[i,j] \\ &= A[\pi_1\mu^{-1}[i], \pi_1\mu^{-1}[j]] \\ &= A_{\pi_1}[\mu^{-1}[i], \mu^{-1}[j]] = A_\mu[\mu^{-1}[i], \mu^{-1}[j]] \\ &= A[\mu\mu^{-1}[i], \mu\mu^{-1}[j]] = A[i,j]. \end{aligned}$$

How to prune with automorphisms?

1. When algorithm `Compare` returns "equal", we record one more automorphism.
2. When branching on the backtracking tree, use known automorphisms for further pruning.
   Example:
   Node N_0: 1|3|567|024
   Children:
   N_1: 1|3|5|67|024
   N_2: 1|3|6|57|024
   N_3: 1|3|7|56|024
   If $g_1 = (24)(56)$ and $g_2 = (04)(57)$ are automorphisms, then prune N_2, since $g_1(N_1) = N_2$ and prune N_3, since $g_2(N_1) = N_3$.

What do we need to compute efficiently in order to prune with automorphisms?

- Store/update information on the automorphisms found so far: if $g_1, g_2, \ldots, g_k$ have been found, store the subgroup S of $Aut(G)$ generated by $g_1, g_2, \ldots, g_k$.
- Quickly determine if partitions $R = q_0|q_1|\cdots|q_{l-1}|u|Q[l] - u|\cdots|$ and $R' = q_0|q_1|\cdots|q_{l-1}|u'|Q[l] - u'|\cdots|$ are equivalent, that is, determine if there exists $g \in S$ such that $g(R) = R'$.

Reviewing some group theory

Definition

A **group** is a set G with operation $*$ such that

1. there exists an identity $I \in G$ such that $g * I = g$ for all $g \in G$, and
2. for all $g \in G$ there exists an inverse $g^{-1} \in G$ such that $g^{-1} * g = I$.

A **subgroup** S of G is a subset $S \subseteq G$ that is a group.

Theorem (Lagrange)

Let G be a finite group. If H is a subgroup of G then

1. *G can be written as $G = g_1 H \cup g_2 H \cup \ldots \cup g_r H$ for some $g_1, g_2, \ldots, g_r \in G$ (where the unions are disjoint)*
2. *$|H|$ divides $|G|$.*

We say that $T = \{g_1, g_2, \ldots, g_r\}$ is a **system of left coset representatives** *or a* **left transversal** *of H in G.*

Permutation groups and automorphism group

Theorem

$Sym(X)$, *the set of all permutations on* X, *is a group under the operation of* composition of functions.

Theorem

$Aut(G)$, *the set of automorphisms of a graph* G, *is a group under the operation of* composition of functions.

Schreier-Sims representation of a permutation group

Let G be a permutation group on $X = \{0, 1, \ldots, n-1\}$, and let

$$\begin{array}{rcl} G_0 & = & \{g \in G : g(0) = 0\} \\ G_1 & = & \{g \in G_0 : g(1) = 1\} \\ & \vdots & \\ G_{n-1} & = & \{g \in G_{n-2} : g(n-1) = n-1\} = I \end{array}$$

$G \supseteq G_0 \supseteq G_1 \supseteq G_2 \cdots \supseteq G_{n-1} = I$ are subgroups.
For all $i \in \{0, 1, 2, \ldots, n-1\}$ (taking $G_{-1} = G$),
let $orb(i) = \{g(i) : g \in G_{i-1}\} = \{x_{i,1}, x_{i,2}, \ldots, x_{i,n_i}\}$ and
$U_i = \{h_{i,1}, h_{i,2}, \ldots, h_{i,n_i}\}$ such that $h_{i,j}(i) = x_{i,j}$.
THEOREM. U_i is a left transversal of G_i in G_{i-1}.
The data structure: $[U_0, U_1, \ldots, U_{n-1}]$ is called the **Schreier-Sims representation** of the group G.
Any $g \in G$ can be uniquely written as $g = h_{0,i_0} * h_{1,i_1} * \cdots * h_{n-1,i_{n-1}}$.

Useful algorithms from Chapter 6

PROCEDURE ENTER$(n, g, [U_0, U_1, \ldots, U_{n-1}])$
INPUT: n, PERMUTATION g, AND $[U_0, U_1, \ldots, U_{n-1}]$, THE SCHREIER-SIMS REPRESENTATION OF G.
OUTPUT: $[U'_0, U'_1, \ldots, U'_{n-1}]$, THE SCHREIER-SIMS REPRESENTATION OF G', THE GROUP GENERATED BY G AND g.

Changing the base: modify the Schreier-Sims representation to work on a base permutation β.
Redefine $G_i = \{g \in G_{i-1} : g(\beta(i)) = \beta(i)\}$.
$[\beta, [U_0, U_1, \ldots, U_{n-1}]]$ is the (modified) Schreier-Sims representation.

PROCEDURE CHANGEBASE$(n,\ [\beta, [U_0, U_1, \ldots, U_{n-1}]],\ \beta')$
INPUT: n, $[\beta, [U_0, U_1, \ldots, U_{n-1}]]$, NEW BASIS β'
OUPUT: $[\beta', [U'_0, U'_1, \ldots, U'_{n-1}]]$

Algorithm 7.10: $\textsc{Cert2}(\mathcal{G}, \vec{G})$ external $\textsc{Canon2}()$

Comment: Set $\vec{G}$ to the identity group with base $\mathbf{I}$.

for $j \leftarrow 0$ to $n-1$ do $\mathcal{U}_j \leftarrow \mathbf{I}$;

$\vec{G} \leftarrow (\mathbf{I}; [\mathcal{U}_0, \mathcal{U}_1, \ldots, \mathcal{U}_{n-1}]$

$P \leftarrow [\{0, 1, 2, \ldots, n-1\}]$

$\textsc{Canon2}(\mathcal{G}, \vec{G}, P)$ (Main algorithm: get $Best$ for canonical adjacency matri

(Next steps: Convert matrix for $Best$ into number (certificate) C:

$C := Num_{Best}(\mathcal{G}) = \min\{Num_\pi(\mathcal{G}) : \pi \in \Pi(G)\}$)

$k \leftarrow 0$; $C \leftarrow 0$

for $j \leftarrow n-1$ downto 1 do

  for $i \leftarrow j-1$ downto 0 do

    if $\{Best[i], Best[j]\} \in \mathcal{E}(\mathcal{G})$ then $C \leftarrow x + 2^k$

    $k \leftarrow k+1$

return (C)

Algorithm 7.9: $\textsc{Canon2}(\mathcal{G}, \vec{G}, P)$ external $\textsc{Refine}()$, $\textsc{Compare}()$, $\textsc{Enter2}()$

$\textsc{Refine}(n, \mathcal{G}, P, Q)$

Find the index l of the first block of Q with $|Q[l]| > 1$

$Res \leftarrow Better$

if $BestExist$ then for $i \leftarrow 0$ to $l-1$ do $\pi_1[i] \leftarrow q_i$, where $Q[i] = \{q_i\}$

$Res \leftarrow \textsc{Compare}(\mathcal{G}, \pi_1, l)$

if Q has n blocks then

if **not** $BestExist$ then

$BestExist \leftarrow \textbf{true}$; for $i \leftarrow 0$ to $n-1$ do $Best[i] \leftarrow q_i$, where $Q[i] = \{q_i\}$

else if $Res = Better$ then $Best \leftarrow \pi_1$

else if $Res = Equal$ then for $i \leftarrow 0$ to $n-1$ do $\pi_2[\pi_1[i]] \leftarrow Best[i]$

$\textsc{Enter2}(\pi_2, \vec{G})$

else if $Res \neq Worse$ then...

(continue in the next page)

(continuing $\textsc{Canon2}()$)

else if $Res \neq Worse$ then
  $ChoicesLeft \leftarrow Q[l]$; $AllChoices \leftarrow Q[l]$
  for $j \leftarrow 0$ to $l-1$ do $R[j] \leftarrow Q[j]$
  for $j \leftarrow l+1$ to $size(Q)$ do $R[j+1] \leftarrow Q[j]$
  while $ChoicesLeft \neq \emptyset$ do
    $u \leftarrow$ any element of $ChoicesLeft$
    $R[l] \leftarrow \{u\}$; $R[l+1] \leftarrow AllChoices \setminus \{u\}$
    $\textsc{Canon2}(\mathcal{G}, \vec{G}, R)$
    for $j \leftarrow 0$ to l do
      $\beta'[j] \leftarrow r$, where $R[j] = \{r\}$
    for each $y \notin \{\beta'[0], \beta'[1], \ldots, \beta'[l]\}$ do
      $j \leftarrow j+1$
      $\beta'[j] \leftarrow y$
    $\textsc{ChangeBase}(n, \vec{G}, \beta')$
    for each $g \in \mathcal{U}_l$ do
      $ChoicesLeft \leftarrow ChoicesLeft \setminus \{g(u)\}$

Backtracking for a certificate for general graphs

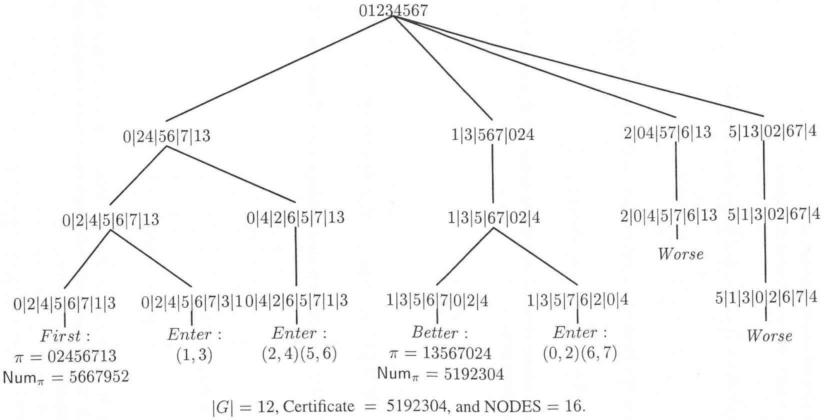


FIGURE 7.3
The state space tree that results from running Algorithm 7.9 on the Graph in Example 7.7.

Using known automorphisms

If we know some or all automorphisms of G we can input the Schreier-Sims representation of the group generated by these automorphisms to the algorithm `Canon2`.
For the previous example, if we input $Aut(G)$, the backtracking tree would have only 10 nodes instead of 16 (see page 273).

Representing other combinatorial objects as coloured graphs

A coloured graph is a graph G plus an ordered partition P of the vertex set. For an ordered partition $P = [P[1], P[2], \ldots, P[l]]$ of $V(G)$, we write $P(v)$ for the index of the block of P that vertex v occurs, i. e. $P(x) = i$ if $x \in P[i]$.

Isomorphism of graphs naturally extends to isomorphism of coloured graphs: an isomorphism of coloured graphs must map vertices of each colour onto vertices of the same colour.

Definition

Two coloured graphs (G_1, P_1) and (G_2, P_2) are **isomorphic** if there is an isomorphism $f : V(G_1) \to V(G_2)$ of G_1 and G_2 such that $P_1(u) = P_2(f(u))$ for all $u \in V(G)$.

Isomorphism of set systems

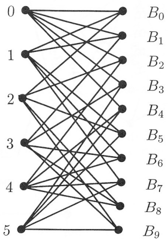
Let $(V, \mathcal{B})$ be a set system (also called incidence structures or hypergraphs)
Define a bipartite graph $G_{V,\mathcal{B}}$ with vertex set $V \cup \mathcal{B}$ and with an edge connecting $x \in V$ to $B \in \mathcal{B}$ if and only if $x \in B$.
This is usually called the point-block incidence graph.

Example 7.9 *A set system and its corresponding graph*

The set system.

$B_0 = \{0, 1, 2\}$
$B_1 = \{0, 1, 3\}$
$B_2 = \{0, 2, 4\}$
$B_3 = \{0, 3, 5\}$
$B_4 = \{0, 4, 5\}$
$B_5 = \{1, 2, 5\}$
$B_6 = \{1, 3, 4\}$
$B_7 = \{1, 4, 5\}$
$B_8 = \{2, 3, 4\}$
$B_9 = \{2, 3, 5\}$

The graph.



Isomorphism of set systems (continued)

Then, $(V_1, \mathcal{B}_1) \sim (V_2, \mathcal{B}_2)$ if and only if $G_{V_1,\mathcal{B}_1} \sim G_{V_2,\mathcal{B}_2}$ with respect to initial partitions $P_1 = [V_1, \mathcal{B}_1]$ and $P_2 = [V_2, \mathcal{B}_2]$, respectively.
We can extract the automorphism group of $(V, \mathcal{B})$ from the automorphism group of $G_{V,\mathcal{B}}$. The automorphism group of $(V, \mathcal{B})$ is the automorphism group of $G_{V,\mathcal{B}}$ restricted to V.

Isomorphism of codes

Definition

A q-ary *code* C of length n is a nonempty subset of Z_q^n; i.e. C is a set of vectors/words x of length n with components $x_i \in Z_q$.

Example: $C = \{0000, 0011, 0201, 0110\} \subseteq Z_3^4$ is a ternary code with 4 words of length 4.

Coding theory is essential for many engineering and computer science applications as well as a topic of purely mathematical interest.

Codes as coloured graphs (see Östergård, *Disc. Appl. Math* 120 (2002))

For a q-ary code $C \subseteq Z_q^n$ defined coloured graph $CG(C)$:

- vertex set: $C \cup \{1, 2, \ldots, n\} \times Z_q$,
- edge set: $\{\{x, (i, x_i)\} : x \in C, i \in \{1, 2, \ldots, n\}\} \cup \{\{(j, a), (j, b)\} : j \in \{1, 2, \ldots, n\}, a, b \in Z_q\}$,
- vertex colouring: $(C, \{1, 2, \ldots, n\} \times Z_q)$

Example: what is the code this graph corresponds to?

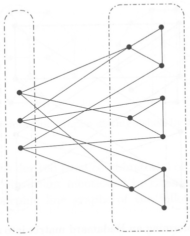


Fig. 3.9. Transforming an unrestricted code into a colored graph

Other combinatorial objects

See Kaski & Östergård book (2007) transforming other combinatorial objects into different coloured graphs:

- set systems/incidence structures using incidence (bipartite) graphs as seen before;
- Steiner triple systems using block intersection graphs (for $v > 15$, the systems is reconstructible from this graph);
- Hadamard matrices,
- other types of codes.

The advantage of using coloured graphs for isomorphism of other structures is to use the power of available tools for graph isomorphism such as "partition refinement+backtracking" algorithms in general (such as $\textsc{Cert1}$ and $\textsc{Cert2}$), and *nauty* software specifically.
However, in some situations, a tailored approach that works directly with the object can be more efficient.