Isomorph-free exhaustive generation
[Ch.4, Kaski & Östergård]

Lucia Moura

Winter 2009

Isomorph-free exhaustive generation

We look at techniques for **exhaustive generating without isomorphs** all objects of certain type.

These techniques have as starting point a backtracking search as we have studied, where we incorporate **isomorph rejection techniques**.

Serves two purposes:

- obtaining desirable isomorph-free list;
- eliminate excessive redundant work.

The main references for these notes are the following book chapters:

- P. Kaski and P. Östergård, Classification Algorithms for Codes and Designs, Springer, 2006.
  Chapter 4: Isomorph-free Exhaustive Generation
- L. Moura and I. Stojmenovic, Backtracking and isomorph-free generation of polyhexes, in *Handbook of Applied Algorithms: Solving Scientific, Engineering, and Practical Problems*, A. Nayak and I. Stojmenovic (eds), 64 pages, John Wiley & Sons, New York, 2008.

Summary of Isomorph-free Exhaustive Generation Techniques using the Search Tree Model

1. **Generate all (or way too many) but record non-isomorphs**
   naïve method: keep only one copy of isomorphic final objects (if only leaves are final, we generate all)
   1. Isomorph rejection via recorded final objects.
   2. Isomorph rejection via canonicity test of final objects.
2. **Generate via an isomorph-free search tree**
   prune isomorphic nodes; above is identical to this if all nodes are final
   1. Isomorph rejection via recorded objects, where we record all intermediate objects found so far.
   2. Orderly generation: Isomorph rejection via canonicity test at each node/intermediate object.
   3. Canonical augmentation: Isomorph rejection via defining canonical extensions of objects, rather than canonical objects.

(The so called "method of homomorphisms" (Laue & others) uses a more algebraic approach, not the search-tree model, and it is not covered here.)

Back to Backtracking... (a few advices by Kaski and Östergård)

- **Incorporate what you know into the algorithm.** Use all combinatorial info available; e.g. fix all that can be fixed.
- **Minimize the number of alternatives in a choice set.** Branch on a variable that minimizes the size of the choice set (e.g. Sudoku: choose to branch on cells that have the least possible number of choices); special case: *constraint propagation* or *forcing*.
- **Abort early if possible.** Pruning via bounding; pruning via checking implied infeasibility (constraints on partial objects).
- **Minimize the amount of work at each recursive call.** The number of nodes is huge, so reduce work at each node; carefully design data structures for the following operations: update d.s. after a choice; rewind d.s. to reflect backtrack; when search returns from a recursive call, need to quickly determine next choice.
- **Keep it simple.** Small loss of efficient is ok for a gain in simplicity; complexity leads to errors.

Definitions and notation

Some notation used in the next Algorithms following Kaski & Östergård:

- The *domain* of a search is a finite set Ω that contains all objects considered in the search.
  e.g. The set of all 0-1 matrices of size 4×4 with entries on 0 or more rows set to value "?".
- A *search tree* is a rooted tree whose nodes are objects in the domain Ω. Two nodes are joined by an edge if and only if they are related by one search step. The root node is the starting point of the search.
- For a node X in a search tree we denote by $C(X)$ the set of child nodes of X. For a non-root node X we denote by $P(X)$ the parent node of X.

Note that a search tree is normally defined only implicitly through the domain Ω, the root node $R \in \Omega$ and the rule $X \to C(X)$.

Four by four 0-1 matrices with exactly two ones in each row and in each column: no isomorph rejection.

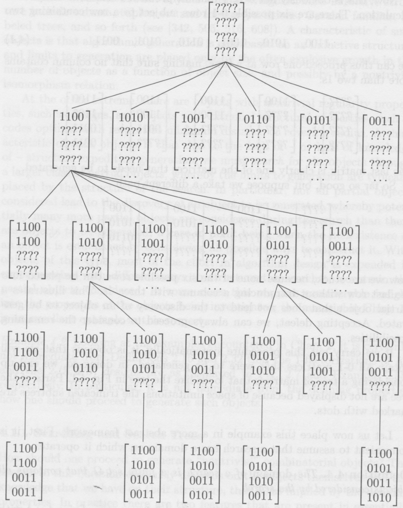


Fig. 4.1. A search tree (truncated in part).

Definitions and notation, continued

- Let G be a group that acts on the search domain Ω. Associate with every $X, Y \in \Omega$ the set

  $$\mathrm{Iso}(X, Y) = \{g \in G : gX = Y\}.$$

  Each element of $\mathrm{Iso}(X, Y)$ is an isomorphism of X onto Y. The objects X and Y are isomorphic if $\mathrm{Iso}(X, Y)$ is non-empty, and we write $X \sim Y$ (or $X \sim_G Y$, to explicitly specify G).

Four by four 0-1 matrices with exactly two ones in each row and in each column:

isomorph rejection

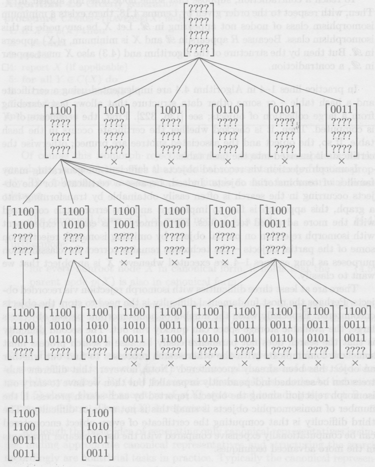


Fig. 4.2. A search tree traversed with isomorph rejection

Recorded objects method: only recording final objects

```
procedure RECORD-FINAL-TRAVERSE($X$:node)
    if COMPLETE($X$) then      (if $X$ is a final object)
        if $\nexists Y \in \mathcal{R}$ such that $X \sim Y$ then
            $\mathcal{R} \leftarrow \mathcal{R} \cup \{X\}$
            output $X$ (optional, since already recorded in $\mathcal{R}$)
    for all $Z \in C(X)$ do
        RECORD-TRAVERSE($Z$)
```

Problems:

- it is naïvely possibly generating the full search tree (lots of isomorphic intermediate nodes).
- A lot of memory required to record all (non-iso) objects.

Recorded objects method: recording all intermediate objects

```
procedure $\textsc{Record-Traverse}(X$:node)
    if $\nexists Y \in \mathcal{R}$ such that $X \sim Y$ then
        $\mathcal{R} \leftarrow \mathcal{R} \cup \{X\}$ (records and checks intermediate objects)
        if $\textsc{complete}(X)$ then output $X$     if $X$ is a final object, output it.
        for all $Z \in C(X)$ do
            $\textsc{Record-Traverse}(Z)$
```

- Solved the first problem: tree has no isomorphic nodes now!
- Second problem is worse: a lot more memory required to record all (non-iso) partial objects.
- In any case, if employing this approach, we need a lot of memory and efficient data structure to search for objects - e.g. hashing table that stores certificate for found objects.

Canonical objects and canonicity testing

Select a **canonical representative** from each isomorphism class of nodes in the search tree.

Denote by ρ the canonical representative map for the action of G on the search domain Ω, that we use to decide weather a node is in canonical form.

The use of ρ eliminates the need to check against previously generated objects. Instead, we only check whether the object of interest is in canonical form, $X = \rho(X)$, and thus we accept it, or is not canonical, $X \neq \rho(X)$, and thus we reject it.

Similarly to checking against recorded objects, we can do canonicity test only on "final nodes" (nodes corresponding to final objects) or at each node.

Canonical object method: canonicity testing for final objects

```
procedure $\textsc{Canrep-Final-Traverse}(X$:node)
    if $\textsc{Complete}(X)$ then if $X = \rho(X)$ then output $X$
    for all $Y \in C(X)$ do
        $\textsc{Canrep-Traverse}(Y)$
```

- Like in $\textsc{Record-Final-Traverse}$, it is naïvely possibly generating the full search tree (lots of isomorphic intermediate nodes).
- Solved the problem of memory and search for recorded isomorphs since no need to record previous objects.

Canonical object method: canonicity testing at each node = Orderly Generation

```
procedure CANREP-TRAVERSE($X$:node)
    if $X = \rho(X)$ then
        Report $X$: if COMPLETE($X$) then output $X$
        for all $Y \in C(X)$ do
            CANREP-TRAVERSE($Y$)
```

Theorem

CANREP-TRAVERSE *reports exactly one node from each isomorphism class of nodes, under the following assumptions:*

- *for every node X, its canonical form $\rho(X)$ is also a node; and*
- *for every non-root node X in canonical form, it holds that the parent node $p(X)$ is also in canonical form.*

Canrep-Traverse is called **orderly generation** due to the typical canonical representative:
an isomorphic object that is extremal in its isomorphism class (largest lexicographically or smallest lexicographically).

The search tree is build so that the most significant parts are completed first.

Orderly generation was introduced independently by Faradzev (1977) and Read (1978).

Orderly generation example (lexicographically larger columns come first)

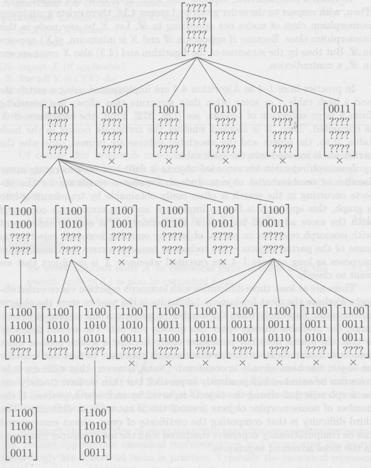


Fig. 4.2. A search tree traversed with isomorph rejection

Canonical augmentation method

- Algorithm by McKay (1998).
- Objects are required to be generated in a canonical way (instead of being canonical).
- We follow Kaski and Östergård's presentation of two variations introduced by McKay:
  - weak canonical augmentation (simplified framework); and
  - canonical augmentation.

  There are situations in which the simplified framework has advantages.

Isomorphic objects may be generated by non-isomorphic paths...

Consider X and Y with $X \sim Y$, and their sequence of ancestors:

X	$p(X)$	$p(p(X))$	$p(p(p(X)))$	$p(p(p(p(p(X)))))$
1100	1100	1100	1100	????
1010	1010	1010	????	????
0101	0101	????	????	????
0011	????	????	????	????

Y	$p(Y)$	$p(p(Y))$	$p(p(p(Y)))$	$p(p(p(p(p(Y)))))$
1100	1100	1100	1100	????
0011	0011	0011	????	????
0110	0110	????	????	????
1001	????	????	????	????

ISO	ISO	NON-ISO!	ISO	ISO

We need a method to specify a canonical way to generate each nonroot node of the search tree.

Defining a weak canonical parent for each nonroot node

Let Ω_{nr} be the union of all orbits of G on Ω that contains a nonroot node of the search tree.

Associate with each $X \in \Omega_{nr}$ a weak canonical parent $w(X)$ with the following property:
for all $X, Y \in \Omega_{nr}$, $X \sim Y$ implies $w(X) \sim w(Y)$.

We define the canonical way to generate X as the (finite) sequence:

$$X, w(X), w(w(X)), w(w(w(X))), \ldots$$

In practice, this is tested one position at a time:
We say that X is generated by *weak canonical augmentation* if

$$p(X) \sim w(X)$$

.

What weak canonical augmentation gives us?

- Let X, Y be isomorphic nodes, generated by canonical augmentation. Then, we get by the definition of canonical augmentation that $w(X) \sim w(Y)$.

What weak canonical augmentation gives us?

- Let X, Y be isomorphic nodes, generated by canonical augmentation. Then, we get by the definition of canonical augmentation that $w(X) \sim w(Y)$.
- Therefore,
$$p(X) \sim w(X) \sim w(Y) \sim p(Y).$$

What weak canonical augmentation gives us?

- Let X, Y be isomorphic nodes, generated by canonical augmentation. Then, we get by the definition of canonical augmentation that $w(X) \sim w(Y)$.
- Therefore,
$$p(X) \sim w(X) \sim w(Y) \sim p(Y).$$
- That is, isomorphic nodes must have isomorphic parents.

What weak canonical augmentation gives us?

- Let X, Y be isomorphic nodes, generated by canonical augmentation. Then, we get by the definition of canonical augmentation that $w(X) \sim w(Y)$.
- Therefore,

$$p(X) \sim w(X) \sim w(Y) \sim p(Y).$$

- That is, isomorphic nodes must have isomorphic parents.
- If isomorphism rejection is applied on parent nodes, then isomorphic nodes must be siblings!!!

What weak canonical augmentation gives us?

- Let X, Y be isomorphic nodes, generated by canonical augmentation. Then, we get by the definition of canonical augmentation that $w(X) \sim w(Y)$.
- Therefore,
$$p(X) \sim w(X) \sim w(Y) \sim p(Y).$$
- That is, isomorphic nodes must have isomorphic parents.
- If isomorphism rejection is applied on parent nodes, then isomorphic nodes must be siblings!!!
- Conclusion:
It is sufficient to apply isomorph rejection on siblings !!!

Obtaining isomorph-free exhaustive generation with weak canonical augmentation

Isomorph-free exhaustive generation relies on the following assumptions:

AW1 Isomorphic nodes have isomorphic children:
for all nodes X, Y, if $X \sim Y$, then for every $Z \in C(X)$ there exists a $W \in C(Y)$ with $Z \sim W$.

AW2 For every nonroot node X, there exists a nonroot node Y such that $X \sim Y$ and $p(Y) \sim w(Y)$.

These assumptions imply that, for every node X of the original tree (before pruning with isomorph rejection), the canonical parent sequence of X is realized on the level of isomorphism classes by a sequence of nodes occurring in the search tree.

Generation by weak canonical augmentation

```
procedure $\textsc{Weak-Canaug-Traverse}(X\colon \text{node})$
    if $\textsc{Complete}(X)$ then output $X$
    $\mathcal{Z} \leftarrow \emptyset$
    for all $Z \in C(X)$ do
        if $p(Z) \sim w(Z)$ then
            $\mathcal{Z} \leftarrow \mathcal{Z} \cup \{Z\}$
    remove isomorphs from $\mathcal{Z}$
    for all $Z \in \mathcal{Z}$ do
        $\textsc{Weak-Canaug-Traverse}(Z)$
```

Theorem

When implemented on a search tree satisfying assumptions AW1 and AW2, WEAK-CANAUG-TRAVERSE *reports exactly one node from every isomorphism class of nodes.*

Defining (strong) canonical augmentation in place of weak

In addition to the requirement that $p(X) \sim w(X)$, we require the augmentation is done in a specific "way".

The ordered pair $(X, p(X))$ contains information on how X was generated by augmenting $p(X)$.

An *augmentation* is defined to be an ordered pair $(X, Z) \in \Omega \times \Omega$.

Two augmentations are *isomorphic*, $(X, Z) \sim (Y, W)$, if and only if there exists a $g \in G$ with $gX = Y$ and $gZ = W$.

Example:

The following augmentations are isomorphic; an isomorphism is $(h, k) = ((12), (1234))$, corresponding to row and column permutations, respectively:

$$\left(\begin{vmatrix} 1100 \\ 1010 \\ 0101 \\ \text{????} \end{vmatrix} , \begin{vmatrix} 1100 \\ 1010 \\ \text{????} \\ \text{????} \end{vmatrix} \right) , \left(\begin{vmatrix} 0101 \\ 0110 \\ 1010 \\ \text{????} \end{vmatrix} , \begin{vmatrix} 0101 \\ 0110 \\ \text{????} \\ \text{????} \end{vmatrix} \right)$$

Defining a canonical parent for each nonroot node

Associate with each $X \in \Omega_{nr}$ a canonical parent $m(X) \in \Omega$ satisfying the following property:
for all $X, Y \in \Omega_{nr}$, $X \sim Y$ implies $(X, m(X)) \sim (Y, m(Y))$.

We define the canonical way to generate X as the (finite) sequence:

$$X, m(X), m(m(X)), m(m(m(X))), \ldots$$

In practice, this is tested one position at a time:
We say that X is generated by *canonical augmentation* if

$$(X, p(X)) \sim (X, m(X)).$$

Note that canonical augmentation implies weak augmentation, but the converse is not true.

What canonical augmentation gives us?

- Let Z, W be isomorphic nodes, generated by canonical augmentation. Then, we get by the definition of canonical augmentation that $(Z, m(Z)) \sim (Z, p(Z))$ and $(W, p(W)) \sim (W, m(W))$.

What canonical augmentation gives us?

- Let Z, W be isomorphic nodes, generated by canonical augmentation. Then, we get by the definition of canonical augmentation that $(Z, m(Z)) \sim (Z, p(Z))$ and $(W, p(W)) \sim (W, m(W))$.
- Therefore, $(Z, p(Z)) \sim (Z, m(X)) \sim (W, m(W)) \sim (W, p(W))$. In particular, $p(Z) \sim p(W)$.

What canonical augmentation gives us?

- Let Z, W be isomorphic nodes, generated by canonical augmentation. Then, we get by the definition of canonical augmentation that $(Z, m(Z)) \sim (Z, p(Z))$ and $(W, p(W)) \sim (W, m(W))$.
- Therefore, $(Z, p(Z)) \sim (Z, m(X)) \sim (W, m(W)) \sim (W, p(W))$. In particular, $p(Z) \sim p(W)$.
- Again, assuming that isomorphism rejection is applied on parent nodes, then isomorphic nodes must be siblings, i.e. $p(Z) = p(W)$.

What canonical augmentation gives us?

- Let Z, W be isomorphic nodes, generated by canonical augmentation. Then, we get by the definition of canonical augmentation that $(Z, m(Z)) \sim (Z, p(Z))$ and $(W, p(W)) \sim (W, m(W))$.
- Therefore, $(Z, p(Z)) \sim (Z, m(X)) \sim (W, m(W)) \sim (W, p(W))$. In particular, $p(Z) \sim p(W)$.
- Again, assuming that isomorphism rejection is applied on parent nodes, then isomorphic nodes must be siblings, i.e. $p(Z) = p(W)$.
- **Moreover:**
  Let $P = p(Z) = p(W)$**. The fact that** $(Z, P) \sim (W, P)$ **implies that there exists an** $a \in Aut(P)$ **with** $aZ = W$**.**

What canonical augmentation gives us?

- Let Z, W be isomorphic nodes, generated by canonical augmentation. Then, we get by the definition of canonical augmentation that $(Z, m(Z)) \sim (Z, p(Z))$ and $(W, p(W)) \sim (W, m(W))$.
- Therefore, $(Z, p(Z)) \sim (Z, m(X)) \sim (W, m(W)) \sim (W, p(W))$. In particular, $p(Z) \sim p(W)$.
- Again, assuming that isomorphism rejection is applied on parent nodes, then isomorphic nodes must be siblings, i.e. $p(Z) = p(W)$.
- **Moreover:**

  Let $P = p(Z) = p(W)$. The fact that $(Z, P) \sim (W, P)$ implies that there exists an $a \in Aut(P)$ with $aZ = W$.
- Conclusion: Any two isomorphic nodes Z, W must be related by an automorphism of their common parent node P !!!

  We can do much more efficient isomorph rejection among siblings.

  Example: if $Aut(P)$ is trivial (identity only), then we don't need to do isomorphism rejection among siblings.

Obtaining isomorph-free exhaustive generation with canonical augmentation

Isomorph-free exhaustive generation relies on the following assumptions:

AA1 Isomorphic nodes have isomorphic children such that an isomorphism applies also to parent nodes:
for all nodes X, Y, if $X \sim Y$, then for every $Z \in C(X)$ there exists a $W \in C(Y)$ with $(Z, X) \sim (W, Y)$.

AA2 For every nonroot node X, there exists a nonroot node Y such that $X \sim Y$ and $(Y, p(Y)) \sim (Y, m(Y))$.

These assumptions are strengthenings of AW1 and AW2.

Generation by canonical augmentation

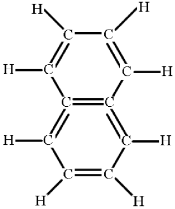
```
procedure $\textsc{Canaug-Traverse}(X\text{: node})$
    if $\textsc{Complete}(X)$ then output $X$
    for all $\mathcal{Z} \in \{C(X) \cap \{aZ : a \in Aut(X)\} : Z \in C(X)\}$ do
        Select any $Z \in \mathcal{Z}$
        if $(Z, p(Z)) \sim (Z, m(Z))$ then
            $\textsc{Canaug-Traverse}(Z)$
```

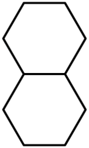
Theorem

When implemented on a search tree satisfying assumptions AA1 and AA2, CANAUG-TRAVERSE *reports exactly one node from every isomorphism class of nodes.*

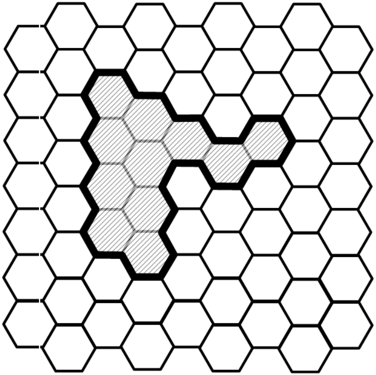
Hexagonal systems (HSs) and computational chemistry

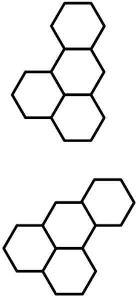
- An h-mino, polyomino or polygonal system consists of h copies of a regular polygon that are connected (two cells are connected by sharing a common edge).
- Polyhexes (or hexagonal systems) correspond to bezonoid hydrocarbons, a class of molecules in organic chemistry composed of carbon and hydrogen atoms.
- Here, we call *hexagonal systems (HSs)* the geometrically planar, simply connected (=no holes) polyhexes.





Hexagonal systems with $h = 11$ and $h = 4$ hexagons





Canonical augmentation for generating hexagonal systems

We will look at two algorithms for the problem of enumerating hexagonal systems, where canonical augmentation led to a breakthrough:

- **BEC algorithm**: Weak canonical augmentation using Boundary Edge Code representation of hexagonal systems.
  G. Caporossi and P. Hansen. Enumeration of polyhex hydrocarbons to $h = 21$. *J. Chem. Inf. Comput. Sci.* **38** (1998), 610–619.
- **LID Algorithm**: Canonical augmentation using Labeled Inner Dual graph representation of fusenes (a class of objects that includes hexagonal systems).
  G. Brinkmann, G. Caporossi and P. Hansen. A constructive enumeration of fusenes and benzenoids. *J. Algorithms* **45** (2002), 155–166.

See also survey by Moura and Stojmenovic (2007).

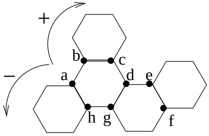
Status of isomorph-free Generation and Enumeration of hexagonal systems

h	$N(h)$	Alg	GE	year
1	1	-	-	
2	1	-	-	
3	3	-	-	
4	7	-	-	
5	22	-	-	
6	81	-	-	
7	331	-	-	
8	1453	-	-	
9	6505	-	-	1965
10	30086	BC	G	1983
11	141229	BC	G	1986
12	669584	BC	G	1988
13	3198256	DAST	G	1989
14	15367577	DAST	G	1990
15	74207910	DAST	G	1990
16	359863778	DAST	G	1990
17	1751594643	CAGE	E	1995
18	8553649747	BEC	G	
19	41892642772	BEC	G	
20	205714411986	BEC	G	
21	1012565172403	BEC	G	1998
22	4994807695197	LID	G	
23	24687124900540	LID	G	
24	122238208783203	LID	G	2002

h	$N(h)$	Alg	GE	year
25	606269126076178	FLM	E	
26	3011552839015720	FLM	E	
27	14980723113884739	FLM	E	
28	74618806326026588	FLM	E	
29	372132473810066270	FLM	E	
30	1857997219686165624	FLM	E	
31	9286641168851598974	FLM	E	
32	46463218416521777176	FLM	E	
33	232686119925419595108	FLM	E	
34	1166321030843201656301	FLM	E	
35	5851000265625801806530	FLM	E	2002

BEC algorithm: BEC code

The BEC algorithm is based on the Boundary Edge Code (BEC code): Select an arbitrary external vertex of degree 3, and follow the boundary of the HS recording the number of boundary edges of each hexagon it traverses. Then, apply circular shifts and/or a reversal, in order to obtain a lexicographically maximum code.
Note that each hexagon can appear 1, 2 or 3 times as digits in the BEC code.

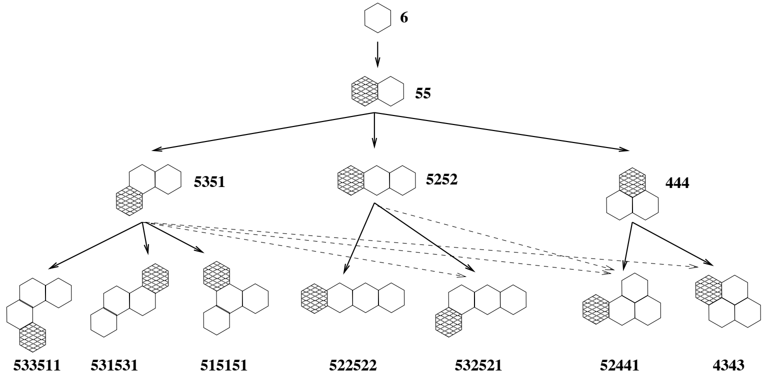


	+	−
a	15115315	51351151
b	51153151	15135115
c	11531515	51513511
d	15315151	15151351
e	53151511	11515135
f	31515115	51151513
g	15151153	35115151
h	51511531	13511515

Definition of canonical parent $w(X)$

Rule for obtaining $w(X)$: the parent of an HS is the one obtained by removing its first hexagon. (This may disconnect the HS, but not for $h \leq 28$)

Example: 5351 below has 6 kids, but it is the canonical parent of only 3.



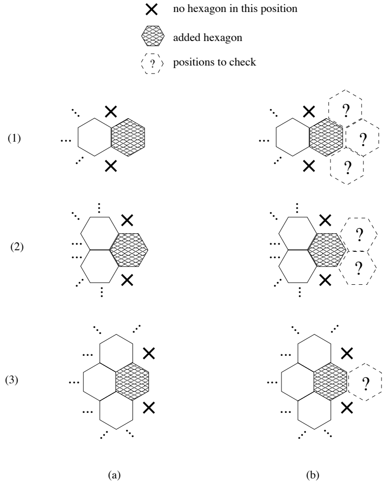
Describing Augmentation

There are three ways in which an hexagon can be added to an HS:

1. A digit $x \geq 3$ in the BEC code corresponding to edges of an hexagon such that one of the edges belong only to this hexagon can be replaced by $a5b$ where $a+b+1=x$ and $a \geq 1$ and $b \geq 1$.
2. A sequence xy in the BEC code with $x \geq 2$ and $y \geq 2$ can be replaced by $(x-1)4(y-1)$.
3. A sequence $x1y$ with $x \geq 2$ and $y \geq 2$ in the BEC code can be replaced by $(x-1)3(y-1)$.

In each of the above cases, we must make sure that the addition of the hexagon does not produce holes.

Three types of augmentation and check for forming holes



BEC Algorithm: Generating valid kids

Procedure GENERATEKIDS generates, from an HS P with j hexagons, its children in the search with $j+1$ hexagons as follows:

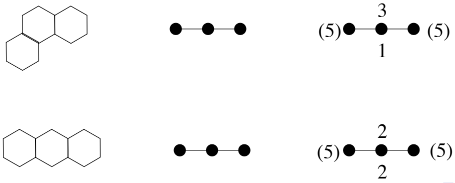
1. **Addition of hexagons:** Any attempt to add a hexagon in the steps below is preceded by a test that guarantees that no holes are created.
    - Add a 5 in every possible way to the BEC code of P.
    - If the BEC code of P does not begin with a 5 then add a 4 in every possible way to the BEC code of P; otherwise, only consider the addition of a 4 adjacent to the initial 5.
    - If the BEC code of P has no 5 and at most two 4's, consider the addition of a 3.
2. **Parenthood validation:**
   For each HS generated in the previous step, verify that its BEC code can begin on the new hexagon. Reject the ones that cannot.

BEC algorithm by weak canonical augmentation

```
procedure $\text{BECG}\textsc{eneration}(P, Pcode, j)$
    if $(j = h)$ then output $P$
    else $S$=$\textsc{GenerateKids}(P, Pcode)$
        Remove isomorph copies from $S$
        for all $(P', Pcode') \in S$ do
            $\text{BECG}\textsc{eneration}(P', Pcode', j+1)$
```

LID Algorithm and Labeled Inner Duals of fusenes

- *Fusenes* are generalizations of HSs that allow for irregular hexagons.
- This algorithm by Brinkmann, Caporossi and Hansen (2002) constructs fusenes and filters them for HSs.
  (Checking weather a fusene fits the hexagonal lattice is not difficult.)
- Define the *inner dual graph* of a fusene as the graph with one vertex for each hexagon, and two vertices are connected if their corresponding hexagons share an edge.
  This graph does not uniquely describe a fusene, but using an appropriate labeling it does (*labeled inner dual*).



Characterization of inner dual graphs

Brinkmann et al.(2002) characterize the graphs that are inner duals of fusenes, which they call id-fusenes. They show that a planar embedded graph G is an id-fusene if and only if

1. G is connected,
2. all bounded faces of G are triangles,
3. all vertices not on the boundary have degree 6, and
4. for all vertices, the total degree, that is the degree plus the number of times it occurs in the boundary cycle of the outer face, is at most 6.

LID algorithm: main steps

1. Generate all non-isomorphic inner dual graphs of fusenes (id-fusenes): algorithm IDFGENERATION uses canonical augmentation.
2. Generate all non-isomorphic labels of inner duals. We have to assign labels, in every possible way, to the vertices that occur more than once on the boundary, so that the sum of the labels plus the degrees of each vertex equals 6. In this process, we must make sure that we do not construct isomorphic labeled inner dual graphs, which can be accomplished by using some isomorphism testing method.

This two step method is very efficient, because...

For two labeled inner dual graphs to be isomorphic, we need that their inner dual graphs be isomorphic. So isomorphic labeled inner dual graphs can only result from automorphisms of the same inner dual graph obtained in step one. Also, very often the inner dual graph has trivial automorphism group (99.9994% of cases for $n = 26$), and so no isomorphism test for its labelings. We focus on the first step!

IDF augmentation

A *boundary segment* of an id-fusene is a set of $l-1$ consecutive edges of the boundary cycle. The vertices of the boundary segment are the end vertices of its edges (there are l of them). For convenience, a single vertex in the boundary cycle is a boundary segment with $l=1$.

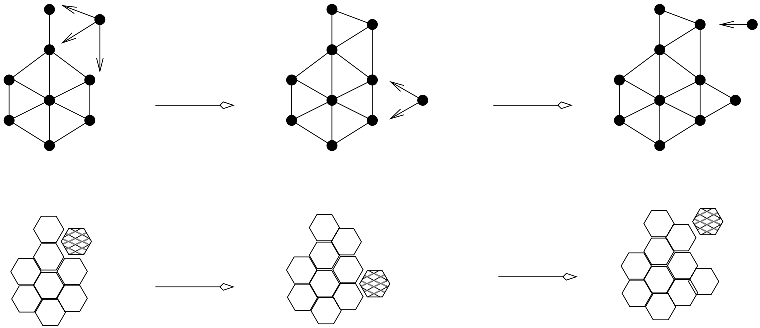
A boundary segment is said to be *augmenting* if the following properties hold: $l \leq 3$, its first and last vertices have total degree at most 5, if $l=1$ its only vertex has total degree at most 4, and if $l=3$ and the middle occurs only once in the boundary, it has total degree 6.

The following shows property AA2 for the canonical augmentation algorithm:

Lemma

All id-fusenes can be constructed from the inner dual of a single hexagon (a single vertex graph) by adding vertices and connecting them to each vertex of an augmenting boundary segment.

Examples of valid augmentations



Definition of canonical parent $m(X)$

McKay (1998) describes a general way of determining parenthood in Algorithm CANAUG-TRAVERSE based on a canonical choice function f. When applied to our case, f is chosen to be a function that takes each id-fusene G to an orbit of vertices under the automorphism group of G that satisfy:

1. $f(G)$ consists of boundary vertices that occur only once in the boundary cycle and have degree at most 3;
2. $f(G)$ is independent on the vertex numbering of G, that is, if Φ is an isomorphism from G to G', then $\Phi(f(G)) = f(G')$.

Now, as described by McKay, graph G is defined to be the parent of graph $G \cup \{v\}$ if and only if $v \in f(G \cup \{v\})$.
The specific f used by Brinkmann, Caporossi and Hansen is a bit technical and omitted here.

IDF Algorithm: Generating valid kids

Procedure $\textsc{GenerateKidsIDF}$ generates, from an id-fusene G with v hexagons, its children in the search tree with $v+1$ hexagons, as follows.

1. **Addition of hexagons:**
   - Compute the orbit of the set of vertices of each augmenting boundary segment of G.
   - Connect the new vertex $n+1$ to the vertices in one representative of each orbit, creating a new potential child graph G' per orbit.
2. **Parenthood validation:**
   For each G' created in the previous step, if $n+1 \in f(G')$ then add G' to S, the set of children of G.

IDF algorithm by canonical augmentation

```
procedure $\text{IDFG}\textsc{eneration}(G, n)$
    if $(n = h)$ then output $G$
    else $S$=$\textsc{GenerateKids}\text{IDF}(G, n)$
        for all $G' \in S$ do
            $\text{IDFG}\textsc{eneration}(G', n + 1)$
```

Recall that unlike weak canonical augmentation, no further isomorphism tests are needed between elements of S.

Distributing the computation

For all these algorithms, since canonical augmentation requires no past memory, the computation can be distributed across independent computers as follows:

Each computer builds the generation tree up to certain level L and then process the generation starting on a node at that level.

For example: if there are p computers, we can ask each computer $i, 0 \leq i \leq p-1$ to handle all nodes k, $1 \leq k \leq L$ with $i = k \mod p$.