Computational Methods to Construct Designs

Lucia Moura
School of Electrical Engineering and Computer Science
University of Ottawa
lucia@eecs.uottawa.ca

Winter 2017

Computational Methods in Design Theory

- The main computation problems involving designs are: **existence, exhaustive generation (or classification)** and **counting**.
  Be aware that the word "enumeration" is often used with different meanings (exhaustive generation or counting).
- **Backtracking** is one of the most important methods for solving these problems.
- **Isomorphism rejection** plays an important role in the efficiency of these methods.

Our presentation follows chapter by Gibbons and Ostergaard, CRC Handbook of combinatorial designs, 2006.

Complexity of Computational Methods

• Most design construction problems seem to be intractable, in the sense that they appear to not have polynomial-time algorithms.

• It also seems unlikely that for generation problems we can have a *polynomial delay* algorithm, i.e. an algorithm that constructs the first design in polynomial time and the next designs within a polynomial delay with respect to the previous one.

• For this reason, we only have available methods that run in worst-case exponential time. Most methods employed fall into two main categories:

- exhaustive search (existence, classification, counting)
- heuristic search (existence (=generation of one object), generation of many distinct objects)

Backtracking

Main ingredients of a backtracking algorithm:

- "Building up feasible solutions one step at a time, covering all possibilities in a systematic fashion." [??]
- If you run long enough, it is guarranted to find an optimal solution (for optimization problems) or garranted to find all feasible solutions (for generation problems).
  WARNING: long enough can be impractical due to combinatorial explosion!
- We need clever pruning techniques, such as:
  - reject partial feasible solutions that cannot lead to a complete solution; and or
  - reject partial feasible solutions that are equivalent to solutions already generated in the search (isomorphism pruning at partial solution level).

Backtracking formulation

We will search for an object in a space $X_1 \times X_2 \times \ldots \times X_n$ where the X_i may or may not be the same.
For all $i \in \{1, 2, \ldots, n\}$, define a boolean-valued *feasibility property*

$$\Pi_i : X_1 \times X_2 \times \ldots \times X_i \to \{\texttt{true}, \texttt{false}\}$$

such that for any $(x_1, x_2, \ldots, x_n) \in X_1 \times X_2 \times \ldots \times X_n$ the following implication holds

$$\Pi_i(x_1, x_2, \ldots, x_i) = \texttt{true} \implies \forall 1 \leq j \leq i, \Pi_j(x_1, x_2, \ldots, x_j) = \texttt{true}.$$

- Existence: find one $(x_1, x_2, \ldots, x_n)$ such that $\Pi_i(x_1, x_2, \ldots, x_n) = \texttt{true}$.
- Classification: find all $(x_1, x_2, \ldots, x_n)$ such that $\Pi_i(x_1, x_2, \ldots, x_n) = \texttt{true}$.
- Counting: find the number of such solutions.

Backtracking: whenever $\Pi_i(x_1, x_2, \ldots, x_i) = \texttt{false}$ this partial solution is not extended.

Backtracking algorithm

Algorithm BacktrackSearch
procedure Search$((x_1, x_2, \ldots, x_i), i)$
begin
  if $i = n$ **then**
    record $(x_1, x_2, \ldots, x_n)$ as a solution
  else for each $x_{i+1} \in X_{i+1}$ **do**
    if $\Pi_{i+1}(x_1, x_2, \ldots, x_{i+1})$ **then**
      Search$((x_1, x_2, \ldots, x_{i+1}), i+1)$
end
main program:
begin
  Search((),0)
end

An execution of the algorithm can be viewed as a *search tree* with one node per recursive call.

Backtracking for BIBDs

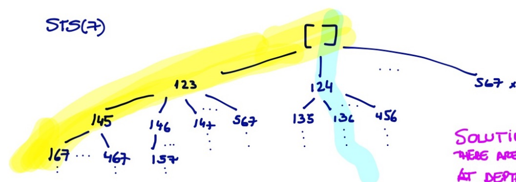
Searching for a BIBD(v, b, r, k, λ) with pointset $P = \{1, 2, \ldots, v\}$ and set of indices of the blocks $B = \{1, 2, \ldots, b\}$.

Block-by-block backtracking:
Use $X_i = \{B \subset P : |B| = k\}$ and
$\Pi_i(x_1, \ldots, x_i) =$
"$|\{x_j : \{a, b\} \subseteq x_j, j \leq i\}| \leq \lambda, \forall \{a, b\} \subseteq P, a \neq b$"

Point-by-point backtracking:
Use $X_i = \{S \subset B : |S| = r\}$ and
$\Pi_i(x_1, \ldots, x_i) =$"$|\{\{j, i\} : |x_j \cap x_i| \neq \lambda, j < i\}| = 0$"

Note: in both cases for generation efficiency, we assume $x_i \leq x_j$ when $i < j$.

Example block-by-block

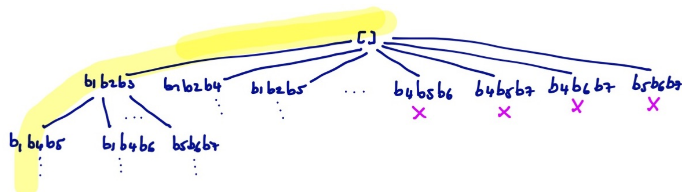


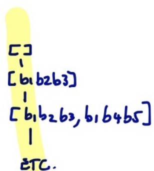
SOLUTIONS:
THERE ARE 35 LEAVES
AT DEPTH b=7

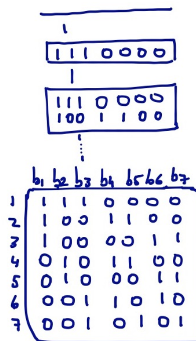
[ ]
[123]
[123, 145]
[123, 145, 167]
[123, 145, 167, 246]
[123, 145, 167, 246, 257]
[123, 145, 167, 246, 257, 356]
[123, 145, 167, 246, 257, 356, 347]
RECORD!

[ ]
[124]
[124, 136]
[124, 136, 157]
[124, 136, 157, 237]
[124, 136, 157, 237, 256]
[124, 136, 157, 237, 256, 345]
CANNOT COMPLETE / BACKTRACK

Example point-by-point







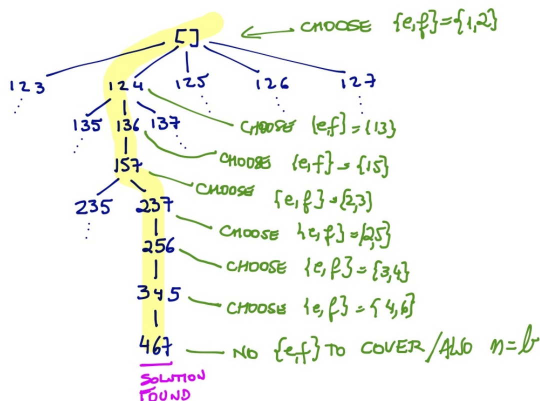
Backtracking for BIBDs

For **block-by-block**, when $\lambda = 1$ we can use the fact that a pair of points must occur in one block and define
$X_i = \{B \subset P : |B| = k, \{e,f\} \subseteq S\}$ where the pair
$\{e,f\} \not\subseteq x_j, \forall j < i$, is fixed using a heuristic that tries to improve performance.

In this case, we do not use $x_i \leq x_j$ for $i < j$ as before, as we traverse blocks in different orders.
Note that the X_i defined above are not uniform throughout the search, but each X_i depends on the partial solution.

Minimum degree heuristic: use $\{e,f\}$ to minimize $|X_i|$, so reducing the degree of the current search tree node (reducing the number of branches out of it).

Example block-by-block using minimum degree heuristic



Block-by-block approach as exact set cover

PROBLEM: Exact Cover
INSTANCE: a collection $\mathcal{S}$ of subsets of $\mathcal{R} = \{0, 1, \ldots, n-1\}$.
QUESTION: Does $\mathcal{S}$ contain an exact cover of $\mathcal{R}$:
does there exist $\mathcal{S}' = \{S_{x_0}, S_{x_1}, \ldots, S_{x_{l-1}}\} \subseteq \mathcal{S}$
such that every element of $\mathcal{R}$ is contained
in exactly one set of $\mathcal{S}'$?

For BIBD(v, b, r, k, λ):

- $R = \{\{x, y\} : \{x, y\} \subseteq \{1, \ldots, v\}, x \neq y\}$
- $S = \{\{\{x, y\} \in K\} : K \subseteq \{1, \ldots, v\}, |K| = k\}$

State-of-the-art algorithm: backtrack using the minimum degree heuristic plus a specific data structure called Knuth's dancing-links.

Point-by-point approach as clique finding in graphs

Definition

Let $G = (V, E)$ be a graph. A set $C \subseteq V$ is a clique if for all $x, y \in C$, $x \neq y$, $\{x, y\} \in E$.

Maximum clique problem: find a clique of maximum cardinality.
Maximum clique exhaustive generation: find all cliques of maximum cardinality.

For BIBD(v, b, r, k, λ):

- $V = \{S \subset B : |S| = r\}$
- $E = \{\{S_1, S_2\} : |S_1 \cap S_2| = \lambda\}$

We can use clique finder algorithms and apply to this graph; for example: program `Cliquer` by Ostergaard.

Improving the search

- **Using isomorph rejection**
  Reject partial solutions that are isomorphic to already generated solutions.
- **Using strong feasibility conditions**
  Example: in point-by-point generation, prune the current branch if one block contains more than k points.
- **Using "look-ahead" techniques**
  Instead of **minimum degree heuristic** one can base their choice on looking ahead on the implications of specific choices. These look-ahead can be turned on and off during the search. Gibbons and Mathon (1995) report on the advantages of look ahead.
- **Branch and bound:** For optimization problems: this is a variation of backtracking that uses **bounding** and often a different exploration order of the search tree (best-first).

The enumeration status for Steiner triple systems

v	number of non-isomorph $STS(v)$
7	1
9	1
13	2
15	80
19	11,084,874,829

The STS(19) were enumerated by Kaski and Ostergaard (2004).

The quest for $STS(19)$

Kaski and Ostergaard's method had the following parts:

1. Enumerate particular partial STS(19) (sets of blocks)
2. Complete the partial STS(19) using block by block minimum degree heuristic.
3. Reject some of the STS(19) so that only one object of each isomorphism class remains.

Part 1 had $14,648$ seeds.

Programming Assignment for Next Week

1. Program a backtracking algorithm for finding STS(v).
2. Apply your program to generate STS(7); you should be able to find the 35 distinct STS(7).
3. Try your program on STS(v) for $v = 9, 13, 15$, and report on your findings.
4. Describe your algorithm and any specifics of your data structures and pruning strategies.
5. For each experiment, report on the number of nodes in the backtracking search tree.

Isomorphism of designs

Two designs are isomorphic if there is a relabeling of the points that "transforms" blockset $\mathcal{A}$ into blockset $\mathcal{B}$.

Definition

Two designs $(X, \mathcal{A})$ and $(Y, \mathcal{B})$, with $|X| = |Y|$, are **isomorphic** if there is a bijection $\alpha : X \to Y$ such that

$$[\{\alpha(x) : x \in A\} : A \in \mathcal{A}] = \mathcal{B}.$$

The bijection α is called an **isomorphism**.

$X = Y = \{1, 2, 3, 4, 5, 6, 7\}$
$\mathcal{A} = \{123, 145, 167, 246, 257, 347, 356\}$
$\mathcal{B} = \{124, 235, 346, 457, 156, 267, 137\}$
$\alpha(1) = 1, \alpha(2) = 2, \alpha(3) = 4, \alpha(4) = 5, \alpha(5) = 6,$
$\alpha(6) = 3, \alpha(7) = 7.$

Automorphism group of a design

An isomorphism of a design to itself is called an **automorphism**. The set of all automorphisms of a design form a group under the operation composition of functions.

$Y = \{1, 2, 3, 4, 5, 6, 7\}$
$\mathcal{B} = \{124, 235, 346, 457, 156, 267, 137\}$
The group of automorphisms of this design is a cyclic group generated by the automorphism $\alpha(i) = (i \bmod 7) + 1$.
Another way to represent this automorphism is using the cycle notation for permutations $\alpha = (1234567)$.
The automorphism group of $(Y, \mathcal{B})$ is

$$G = \{\alpha^0, \alpha^1, \alpha^2, \alpha^3, \alpha^4, \alpha^5, \alpha^6\}.$$

$\alpha^0 = (1)(2)(3)(4)(5)(6)(7)$ is the identity permutation,
$\alpha^1 = (1234567), \alpha^2 = (1357246), \ldots, \alpha^6 = (1765432)$

Design Isomorphism as Coloured-Graph Isomorphism

Definition

Given a design $D = (V, \mathcal{B})$ where $V = \{x_1, \ldots, x_v\}$ and $\mathcal{B} = \{B_1, \ldots, B_b\}$, define $G(D)$ to be a graph with vertex set $\{x_1, x_2, \ldots, v_v, B_1, B_2, \ldots, B_b\}$ with the x_i vertices having one colour and the B_i vertices having a second color, and edgeset $\{\{x_i, B_j\} : x_i \in B_j\}$. The graph $G(D)$ is the *Levi graph* of D.

Proposition

1. Designs D_1 and D_2 are isomorphic if and only if graphs $G(D_1)$ and $G(D_2)$ are isomorphic (note the graph isomorphism is required to preserve colours).
2. The automorphism group of a simple design D is isomorphic to the automorphism group of the graph $G(D)$.

Design Isomorphism as Coloured-Graph Isomorphism

Algorithmic consequences

1. To test if two designs are isomorphic we can test if their Levi graphs are isomorphic.
2. To compute the automorphism group of a design, we can compute the automorphism group of its Levi graph.

Both tasks can be done using the `nauty` software developed by Brendan McKay that returns a certificate for isomorphism of coloured graphs and the automorphism group of a graph.
`http://cs.anu.edu.au/~bdm/nauty`

Computing Isomorphism

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.

Isomorphism Invariants: examples

If two objects disagree on the invariant, then the objects are NOT isomorphic; the converse is not true.
They are useful as quick checks to determine two objects are not isomorphic.
Example of invariant: Number of triangles in a graph.

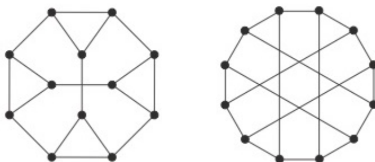


Fig. 3.11. Two nonisomorphic graphs distinguished by the number of triangles

(picture from book by Kaski and Ostergaard 2006)

Isomorphism Invariants: Steiner triple systems

- Size of automorphism group: for the STS(15) this varies between 2 and 20160.
- Chromatic index of an STS: chromatic number of its block intersection graph; for STS(15) this is 7, 8 or 9.
- Number of parallel classes: The number of parallel classes in an STS(15) varies from 0 to 56. These corresponds to the number of independent sets of the block intersection graph that has size $v/3$.
- Number of Pasch configurations: 4 triples on 6 elements of the form $\{a, b, c\}$, $\{a, d, e\}$, $\{b, e, f\}$, $\{c, d, f\}$. For STS(15) the number of Pasch configurations varies from 0 to 105.
- Block intersection graphs: this is in general an invariant, but for STS(15) it can distinguish all 80 non-isomorphic ones.

See master's thesis by Sally Shaul Kazin (2005) https://getd.libs.uga.edu/pdfs/kazin_sally_s_201205_ma.pdf

Isomorphism Invariants: Steiner triple systems

The following non-isomorphic STS(15) have both 12 Pasch configurations:

```
abc ade afg ahi ajk      abc ade afg ahi ajk
alm ano bdf beh bgj      alm ano bdf beh bgj
bik bln bmo cdg cel      bil bkn bmo cdg cef
cfh cij cmn cko dim      cio chl cjn ckm dij .
dkn hjn dho djl efn      dmn dho dkl hjm fim
gin eio ejm egk fil      egm ein eko ejl fjo
fjo fkm ghm glo hkl      fhk fln ghn gik glo
```

A more powerful invariant is the multiset that contains for each block B, the number of Pasch configurations that uses block B. This invariant distinguishes the two systems above.

(Example from book by Kaski and Ostergaard 2006)

Isomorphism Certificates: examples

Two objects are isomorphic if and only if they agree on the certificate.

- To check isomorphism of trees, a certificate can be build in polynomial time.
- For general graphs, we do not have that and the most common methods have exponential worst-case time. One quite fast method in this genre is the `nauty` software by Brendan McKay.
- A certificate for simple graphs is based on its adjacency matrix. Consider the binary $(n(n-1)/2)$-tuple obtained from listing the entries above the diagonal of its incidence matrix from top to bottom; among all graphs isomorphic to G, get the tuple that is the lexicographical largest to be the certificate of G.

Certificate for graphs

Example: the 11 non-isomorhic graphs on 4 vertices have the following certificates:

$$\begin{bmatrix}0000\\0000\\0000\\0000\end{bmatrix},\begin{bmatrix}0100\\1000\\0000\\0000\end{bmatrix},\begin{bmatrix}0100\\1000\\0001\\0010\end{bmatrix},\begin{bmatrix}0110\\1000\\1000\\0000\end{bmatrix},\begin{bmatrix}0110\\1001\\1000\\0100\end{bmatrix},\begin{bmatrix}0111\\1000\\1000\\1000\end{bmatrix},$$

$$\begin{bmatrix}0110\\1010\\1100\\0000\end{bmatrix},\begin{bmatrix}0110\\1001\\1001\\0110\end{bmatrix},\begin{bmatrix}0111\\1010\\1100\\1000\end{bmatrix},\begin{bmatrix}0111\\1011\\1100\\1100\end{bmatrix},\begin{bmatrix}0111\\1011\\1101\\1110\end{bmatrix}.$$

Canonical representatives: examples

Two objects are isomorphic if and only if they have the same canonical form.

Keep only the canonical representative of one's isomorphism class; in other words, if the object is not canonical, reject it.

This notion is related but not quite the same as the notion of certificates. For each isomorphism class, we can define an unique object that is the *canonical* object for the isomorphim class (canonical representative).

Examples:

- graphs: pick the graph whose incidence matrix gives the certificate in the previous example.
- designs: for a design, consider the point-block incidence matrix; in each isomorphism class, select the canonical representative to be the one that is lexicographical largest (or alternatively, smallest).

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.
   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.
   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.

Not covered: "canonical augmentation" (McKay 1998) and "method of homomorphisms" (Laue & others).

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 $X \to C(X)$.

4x4 0-1 matrices with 2 ones in each row and column: no isomorph rejection.

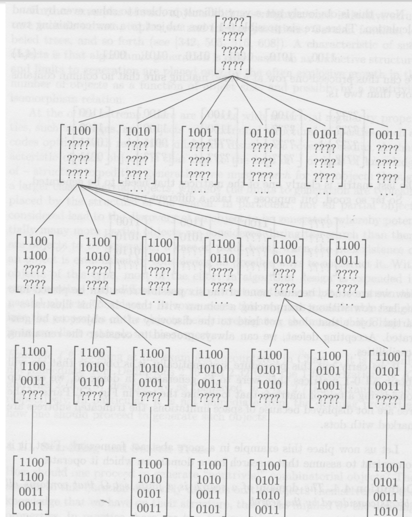


Fig. 4.1. A search tree (truncated in part)

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).

4x4 0-1 matrices with 2 ones in each row and column: isomorph rejection

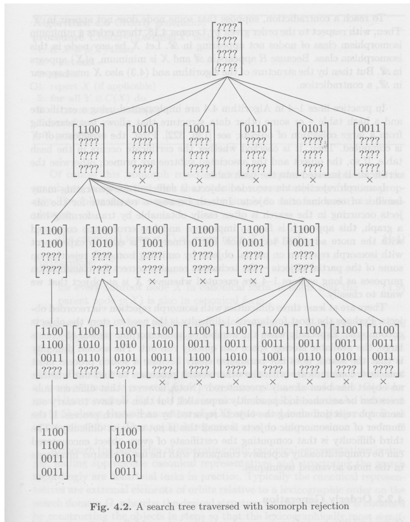


Fig. 4.2. A search tree traversed with isomorph rejection

Recorded objects method: only recording final objects

procedure $\textsc{Record-Final-Traverse}(X\text{:node})$
  if $\textsc{complete}(X)$ 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
    $\textsc{Record-Traverse}(Z)$

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 $\textsc{Record-Traverse}(X\text{:node})$
  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 i
    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 CANREP-FINAL-TRAVERSE(X:node)
  if COMPLETE(X) then if $X = \rho(X)$ then output X
  for all $Y \in C(X)$ do
    CANREP-TRAVERSE(Y)

- Like in 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)

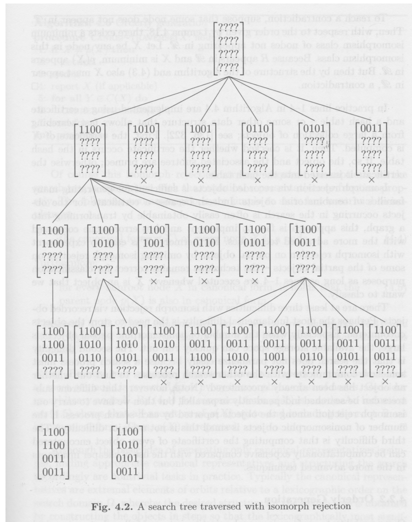


Fig. 4.2. A search tree traversed with isomorph rejection

Orderly generation for BIBDs using the point-by-point approach

For each BIBD(v, b, r, k, λ) we need to generate an incidence matrix, which is a 0-1 matrix such that:

- There are r 1's per row.
- There are k 1's per column.
- The inner product between any two distinct rows is λ.

Orderly generation considers the lexicographical order of $s \times t$ matrices $A = (a_{ij})$ as the lexicographical order of a corresponding st-tuples:
$w(A) = (a_{11}, a_{12}, \ldots, a_{1t}, a_{21}, a_{22}, \ldots, , a_{2t}, \ldots, a_{s1}, a_{s2}, \ldots, , a_{st})$
Two matrices are **isomorphic** if one can be obtained from the other by permuting the rows and the columns.
We say that a matrix is **canonical** (a canonical rep of its isomorphism class) if it is the lexicographic maximum of matrices in its isomorphism class.

Orderly generation for BIBDs (contd)

Why the following conditions for CANREP-TRAVERSE to work are satisfied in this case?

- 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.

Theorem

Let A be a canonical 01 matrix of size $s \times t$. Then the submatrix $A[\{1, 2, \ldots, i\}, .]$ (first i rows of A) is canonical for any $1 \leq i \leq s$.

Orderly generation for BIBDs: example

Generate canonical incidence matrices of the designs.

Example 6.2. Figure 6.1 shows the first five levels of the search tree for 2-(7, 3, 2) designs. Only canonical matrices are displayed.

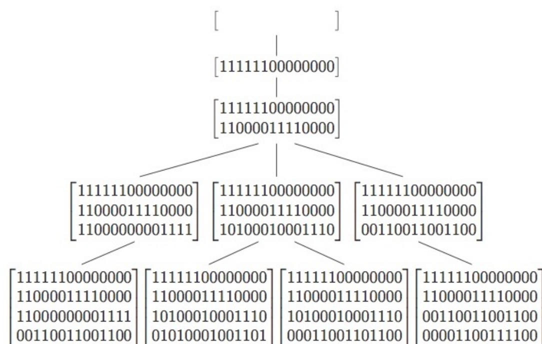


Fig. 6.1. A search tree for 2-(7, 3, 2) designs (truncated to first five levels)

Canonicity testing for BIBDs

A canonicty testing checks if a matrix A is the lexicographical maximum of its isomorphism class.

This is done via **backtracking**, but extensive pruning is possible.

For more details see Chapter 6 of Kaski and Ostergaard 2006.

One thing to note is that we do not need to go through every row and column permutation. Once a row permutation is fixed we can obtain the lexicographical largest column permutation by sorting the columns in decreasing lexicographical order.

Example: Consider a matrix A' whose row permutation $[2, 4, 3, 1]$ gives matrix A below. Matrix $\vec{A}$ is the corresponding column sorted matrix obtained from A:

$$\mathbf{A} = \begin{bmatrix} 11000011110000 \\ 01010001001101 \\ 10100010001110 \\ 11111100000000 \end{bmatrix}, \quad \vec{\mathbf{A}} = \begin{bmatrix} 11111100000000 \\ 11000011110000 \\ 00110011001100 \\ 10100000101011 \end{bmatrix}$$

Question: is the original matrix A' canonical?

To learn more about orderly generation of BIBDs using the point-by-point approach, see:

- Denny and Gibbons, "Case studies and new results in combinatorial enumeration", *J Combinatorial Designs* **8** (2000), 239-260.
- Denny, Search and enumeration techniques for incidence structures, *Master's thesis*, University of Auckland, 1998.

Classification of BIBDs using Block by Block approach

The following approach has been used successfully:

1. Create a set of *seed* subsystems (each possible design must contain at least one of these seeds).
2. Classify the seeds up to isomorphism.
3. Use another procedure to extend the seeds to every possible full design.
4. Remove isomorphs from the list of designs obtained.

We will exemplify how this approach was used by Kaski and Ostergaard to classify all the (over 11 billion) STS(19).

STS(19): the seeds

Example 6.20. Any block $\{x, y, z\}$ of an STS(19) has a nonempty intersection with $1 + 3(r - 1) = 25$ blocks. Disregarding the block $\{x, y, z\}$, the blocks incident with each of the points x, y, z form edge sets of three pairwise edge-disjoint 1-factors on the remaining 16 points. Thus, up to isomorphism such a set of 25 blocks can be described by an incidence matrix of the form in Fig. 6.4, where the matrices $\mathbf{F}_1$ and $\mathbf{F}_2$ specify two 1-factors.

1	11111111	00000000	00000000
1	00000000	11111111	00000000
1	00000000	00000000	11111111
0	10000000		
0	10000000		
0	01000000		
0	01000000		
0	00100000		
0	00100000		
0	00010000		
0	00010000	$\mathbf{F}_1$	$\mathbf{F}_2$
0	00001000		
0	00001000		
0	00000100		
0	00000100		
0	00000010		
0	00000010		
0	00000001		
0	00000001		

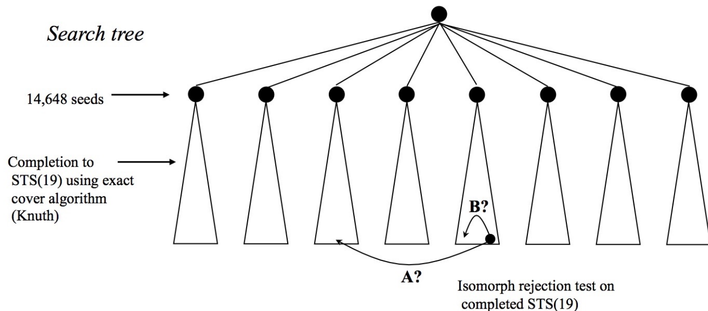
Fig. 6.4. Structure of seeds for STS(19)

STS(19): the seeds

10000000	10000000	10000000	10000000	10000000	10000000	10000000
01000000	01000000	01000000	01000000	01000000	01000000	01000000
10000000	10000000	10000000	10000000	10000000	10000000	10000000
01000000	01000000	01000000	01000000	00100000	00100000	00100000
00100000	00100000	00100000	00100000	01000000	01000000	01000000
00010000	00010000	00010000	00010000	00100000	00010000	00010000
00100000	00100000	00100000	00100000	00010000	00100000	00100000
00010000	00010000	00001000	00001000	00001000	00010000	00001000
00001000	00001000	00010000	00010000	00010000	00001000	00010000
00000100	00000100	00001000	00000100	00000100	00000100	00000100
00001000	00001000	00000100	00001000	00001000	00001000	00001000
00000100	00000010	00000010	00000010	00000010	00000010	00000010
00000010	00000100	00000100	00000100	00000100	00000100	00000100
00000001	00000001	00000001	00000001	00000001	00000001	00000001
00000010	00000010	00000010	00000010	00000010	00000010	00000010
00000001	00000001	00000001	00000001	00000001	00000001	00000001

Fig. 6.5. Possible choices for the $\mathbf{F}_1$ matrix in Fig. 6.4

Combining these choices of F_1 with all the choices of F_2 there are $14,648$ nonisomorphic 25-blocks seeds.

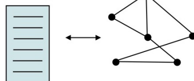


(slide by Peter Gibbons 2005)

Steiner triple systems of order 19

Isomorph rejection

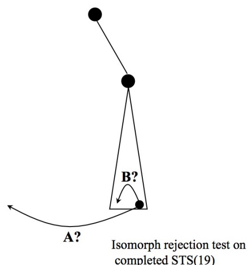
Represent STS(19) by block intersection graph



Use *nauty* to generate graph's canonical labelling

Note: partial configurations not isomorph checked here

Note: In orderly algorithm only canonical partial configurations extended at each level



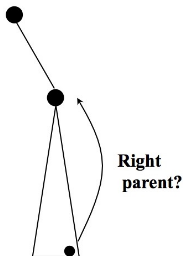
Isomorph rejection test on completed STS(19)

(slide by Peter Gibbons 2005)

Steiner triple systems of order 19

Canonical augmentation

Insist that designated block {0,1,2} lies in first orbit in canonical ordering of orbits

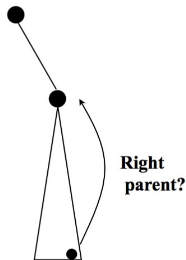


(slide by Peter Gibbons 2005)

Steiner triple systems of order 19

Pasch configurations

0 2 3
0 4 5
1 2 4
1 3 5



Provides invariant

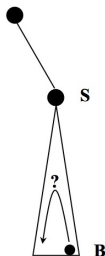
Designated block must lie in maximum number of Pasch configurations

(slide by Peter Gibbons 2005)

Steiner triple systems of order 19

"Family" isomorphism check

For all f in Aut(S) $B \le f(B)$



Seed groups pre-computed and stored if ≤ 200

If > 200, use hash table isomorph rejection strategy

(slide by Peter Gibbons 2005)

Steiner triple systems of order 19

Results

11,084,874,829 pairwise non-isomorphic STS(19)

1,348,410,350,618,155,344,199,680,000 distinct STS(19)

2 years of CPU time

Distributed on ~ 80 workstations

(slide by Peter Gibbons 2005)

Nonexistence of Projective Planes of Order 10

The most important achievement in the classification of designs was the exhaustive search for a projective plane of order 10, that determined it does not exist.
This would be equivalent to a symmetric BIBD$(111, 11, 1)$.

This herculean task was performed by:
LAM, THIEL AND SWIERCZ, The nonexistence of projective planes of order 10, *Canadian J. Mathematics* **41** (1989), 1117-1123.

This was more recently independently verified by Dominique Roy in his master's thesis at Carleton (2010).

https://curve.carleton.ca/system/files/etd/cf019cde-3f3e-44f6-9472-66c83299cee2/etd_pdf/3937a3a053481c7f893b40477c790e62/roy-confirmationofthenonexistenceofaprojective.pdf

Dominique will give us an invited lecture next week on the topic.

Programming exercise for next class

For next week, your assignment will be to use some of the isomorphism rejection techniques studied to list all non-isomorphic STS(v) for the first few values.

The number of nonisomorphic STS(v) for $v = 7, 9, 13, 15$ are $1, 1, 2$ and 80 respectively.

Report on your methods and results.

Heuristic search

Characteristics

- The state space is not fully explored.
- Randomization is often employed.
- There is a concept of neighbourhood search.
- **Heuristics** are applied to explore the solutions.
  The word "heuristics" means "serving or helping to find or discover" or "proceeding by trial and error".

Heuristic search approaches

The problem should be transformed so that a profit function is maximized.

For search problems, we create a function that reflects progress towards a feasible solution.

Types of heuristic search:

- **Hill climbing:** go up the hill continuously until cannot increase profit.
  (can get stuck on local optima)
- **Simulated annealing:** can go down hill according to a controled probability.
- **Tabu Search:** can go downhill to scape a local maximum; keeps tabu list to avoid cycling.
- **Genetic algorithms:** population of solutions is recombined to obtain "offsprings" with higher profit.
- etc.

Searching for Steiner Triple Systems

As Steiner triple systems are BIBD$(v, 3, 1)$, we have that the point replication number is $r = \frac{v-1}{2}$ and the number of blocks is $b = \frac{v(v-1)}{6}$
Recall that for Steiner triple systems, the necessary conditions for existence are also suficient.

Theorem

$$\exists STS(v) \iff v \equiv 1, 3 \pmod 6)$$

So, there exists an $STS(v)$ for $v = 1, 3, 7, 9, 13, 15, 19, 21, 25, \ldots$
The fact that we know a few constructions for Steiner triple systems, does not mean we can generate a good range of non-isomorphic triple systems. Exhaustive search limits us in the sizes of v for which we can compute all Steiner triple systems. Heuristic search can help us to build many distinct triple systems for the same order v through exploring the search space.

Searching for Steiner Triple Systems

A partial Steiner triple system consists of a set of triples $\mathcal{B}$ with each pair of points appearing in at most one $B_i \in \mathcal{B}$. Then, we can formulate the search problem as follows.

PROBLEM: CONSTRUCT A STEINER TRIPLE SYSTEM
INSTANCE: v SUCH THAT $v \equiv 1, 3 \pmod 6$
FIND: MAXIMIZE $|\mathcal{B}|$
SUBJECT TO: $([1, v], \mathcal{B})$ IS A PARTIAL STEINER TRIPLE SYSTEM

The **universe** $\mathcal{X}$ is the set of all sets of blocks $\mathcal{B}$, such that $([1, v], \mathcal{B})$ is a partial Steiner triple system.

An **optimal solution** is any feasible solution with $|\mathcal{B}| = \frac{v(v-1)}{6}$.

Stinson's hill-climbing algorithm for STSs

```
Algorithm Stinson's Algorithm(v)
    Numblocks ← 0
    V ← {1, 2, ... v}
    B ← ∅
    While (Numblocks < v(v−1)/2) do { SWITCH()}
    output (V, B)
```

Switch will either add a new block or substitute one existing block by a new block.

There is no guarantee that the algorithm will ever terminate, but if the choices done by heuristic $\textsc{Switch}()$ are random, it seems in practice that the algorithm always terminates successfully and runs quickly.

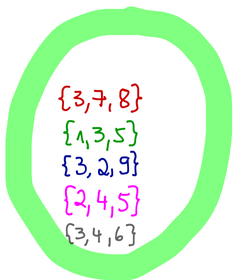
Stinson's hill-climbing for STSs: Switch Algorithm

Definition

A point x is said to be a **live point** in $([1, v], \mathcal{B})$ if $r_x < \frac{v-1}{2}$.
A pair $\{x, y\}$ is said to be a **live pair** in $([1, v], \mathcal{B})$ if there exists no $B \in \mathcal{B}$ with $\{x, y\} \subseteq B$

```
Algorithm SWITCH()
    Choose a random live point x.
    Choose random y, z such that
      {x, y} and {x, z} are live pairs.
    If ({y, z} is a live pair) then
       B ← B ∪ {{x, y, z}}
       Numblocks ← Numblocks +1
    else
        Let {w, y, z} ∈ B be the block containing {y, z}
        B ← B ∪ {{x, y, z}} \ {{w, y, z}}
```

Example



LIVE POINTS: 1, 2, ~~3~~, 4, 5, 6, 7, 8, 9

LIVE PAIRS:

1: 2, 4, 6, 7, 8, 9
2: 1, 6, 7, 8
3: –
4: 1, 7, 8, 9
5: 6, 7, 8, 9
6: 1, 2, 5, 7, 8, 9
7: 1, 2, 4, 5, 6, 9
8: 1, 2, 4, 5, 6, 9
9: 1, 4, 5, 6, 7, 8

Implementing `Switch` in constant time

The presentation was adapted from Kreher and Stinson (1998). We changed the initialization of live pairs to simplify comprehension as well as separated the basic operations on the data structures from the main code that uses it, making it more in line with the object oriented approach.

We keep a **list of live points** using the following data structure which allows insertions and deletions in constant time:

```
int NumLivePoints; int NumLivePoints[1..v];
int IndexLivePoints[1..v];
```

A list of **live pairs** can also allow constant t. insertion and deletion:

```
int NumLivePairs[1..v], int LivePairs[1..v][1..v-1],
int IndexLivePairs[1..v][1..v]
```

Block storage and detection if a pair is live can be achived using `Other[x][y]`, with `Other[x][y]=z` if $\{x, y, z\}$ is a block and `Other[x][y]=0` if pair $\{x, y\}$ is live.

Basic Operations: Live Points

```
Global: int NumLivePoints, int NumLivePoints[1..v], int IndexLivePoints[1..v];

InitializeLivePoints() {
  NumLivePoints=0;
  for x=1 to v do IndexLivePoint[x]=0;
  }

AddLivePoint(x) {
    NumLivePoints++; pos= NumLivePoints;
    LivePoints[pos]=x;
    IndexLivePoints[x]=pos;
}

RemoveLivePoint(x) {
   pos = IndexLivePoints[x];
   IndexLivePoints[x]=0; // indicates x is dead point
   // swap last element to occupy the space freed at pos:
   lastElement= LivePoints[NumLivePoints];
   LivePoints[pos]=lastElement;
   NumLivePoints--;
}
Initialize(v) {
     InitializeLivePoints();
     InitializeLivePairs();
     for x = 1 to v do {
          AddtoLivePoints(x);
          for y=1 to v do
               if (x!=y) AddToLivePairs(x,y);
               Other[x,y]=0;
         }
```

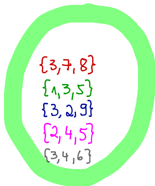
Basic Operations: Live Pairs

```
Global variables: int NumLivePairs[1..v], LivePairs[1..v][1..v-1], IndexLivePairs[1..v][1..v]

InitializeLivePairs() {
   NumLivePairs=0;
   for x=1 to v do
        for y=1 to v do
             IndexLivePairs[x][y]=0;
}

AddLivePair(x,y) {
    If (NumLivePairs[x]=0) AddLivePoint(x);
    NumLivePairs[x]++;
    int pos= NumLivePairs[x];
    LivePairs[x][pos]=y;
    IndexLivePairs[x][y]=pos;
}

RemoveLivePair(x,y) {
   pos = IndexLivePairs[x][y];
   IndexLivePairs[x][y]=0; // indicates {x,y} is dead pair
   // swap the element to occupy the space freed at position pos
   lastpos=NumLivePairs[x];
   lastElement= LivePairs[x][lastpos];
   LivePairs[x][pos]=lastElement;
   NumLivePairs[x]--;
   If (NumLivePairs[x]=0) RemoveLivePoint(x);
}
```

Initializing arrays

```
RevisedSwitch() {
   r=randomInt(1,NumLivePoints);
   x = LivePoints[r];
   s=random(1,NumLivePairs[x]);
   t=random(1,NumLivePairs[x]-1); if (t>=s) t++; // {s,t} random distinct pair
   y = LivePairs[x][s]; z = LivePairs[x][t]
   if (Other[y,z]=0) { // add block {x,y,z}
      Other[x,y]=Other[y,x]=z; Other[x,z]=Other[z,x]=y; Other[y,z]=Other[z,y]=x;
      RemoveLivePair(x,y); RemoveLivePair(y,x); RemoveLivePair(x,z);
      RemoveLivePair(z,x); RemoveLivePair(y,z);RemoveLivePair(z,y);
      Numblocks++;
   }
   else { // exchange block: add {x,y,z} remove {w,y,z}
      Other[x,y]=Other[y,x]=z; Other[x,z]=Other[z,x]=y; Other[y,z]=Other[z,y]=x;
      RemoveLivePair(x,y); RemoveLivePair(y,x); RemoveLivePair(x,z); RemoveLivePair(z,x);
      Other[w,y]=Other[y,w]=Other[w,z]=Other{z,w]=0;
      AddLivePair(w,y); AddLivePair(y,w);AddLivePair(w,z); AddLivePair(z,w);
}}
RevisedStinsonAlgorithm(v) {
        NumBlocks = 0;
        Initialize(v);
        While (NumBlocks < v(v-1)/6) do
             RevisedSwitch();
        // construct blocks from array Other[,]
        for x=1 to v do
           for y=x+1 to v do {
               z=Other[x,y];
               if z>y then Blocks=Blocks U {{x,y,z}}
           }
        return Blocks
```

Example of data structure



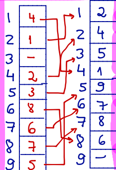
LIVE POINTS: 1, 2, 3, 4, 5, 6, 7, 8, 9

LIVE PAIRS:

1: 2, 4, 6, 7, 8, 9
2: 1, 6, 7, 8
3: –
4: 1, 7, 8, 9
5: 6, 7, 8, 9
6: 1, 2, 5, 7, 8, 9
7: 1, 2, 4, 5, 6, 9
8: 1, 2, 4, 5, 6, 9
9: 1, 4, 5, 6, 7, 8

	1	2	3	4	5	6	7	8	9
1	0	0	5	0	3	0	0	0	0
2	0	0	9	5	4	0	0	0	3
3	5	9	0	6	1	4	8	7	2
4	0	5	6	0	2	3	0	0	0
5	3	4	1	2	0	0	0	0	0
6	0	0	4	3	0	0	0	0	0
7	0	0	8	0	0	0	0	3	0
8	0	0	7	0	0	0	3	0	0
9	0	3	2	0	0	0	0	0	0

IndexLivePoints LivePoints



IndexLivePair

	1	2	3	4	5	6	7	8	9
1	0	1	0	4	0	5	2	3	6
2	3	0	0	0	0	1	2	4	0
3	0	0	0	0	0	0	0	0	0
4	4	0	0	0	0	0	2	1	3
5	0	0	0	0	0	4	3	1	2
6	1	2	0	0	5	0	7	3	4
7	4	1	0	6	2	3	0	0	5
8	4	5	0	6	3	2	0	0	1
9	3	0	0	3	2	6	1	4	0

LivePair

	1	2	3	4	5	6	7	8
1	2	7	8	4	6	9		
2	6	7	1	8				
3								
4	8	7	9	1				
5	8	9	7	6				
6	1	2	8	9	5	7		
7	2	5	6	1	9	4		
8	9	6	5	1	2	4		
9	7	5	4	8	1	6		

NumLivePairs

1	6
2	4
3	0
4	4
5	4
6	6
7	6
8	6
9	6

References

1. P.B. Gibbons and P. J. Ostergaard, Computational Methods in Design Theory, *CRC Handbook of Combinatorial Designs*, Colbourn and Dinitz (eds), 2nd edition, 2006.
2. W. Wallis, Computational and constructive design theory, Springer, 1996.
3. P. Kaski and P.J. Ostergaard, Classification algorithms for codes and designs, Springer, 2006.
4. D. Kreher and D. Stinson, Combinatorial algorithms: generation, enumeration and search, CRC, 1998.