Enhancements to the Perfect Matching-based Tree Algorithm for Generating Architectures

Technical Report UIUC-ESDL-2017-02

Daniel R. Herber*, James T. Allison†
Engineering System Design Lab
University of Illinois at Urbana-Champaign

December 10, 2017

Abstract

In this report, a number of enhancements to the perfect matching-based tree algorithm for generating the set of unique, feasible architectures are discussed. The original algorithm was developed to generate a set of colored graphs covering the graph structure space defined by (C, R, P) and various additional network structure constraints. The proposed enhancements either more efficiently cover the same graph structure space or allow additional network structure constraints to be defined. The seven enhancements in this report are replicate ordering, avoiding loops, avoiding multi-edges, avoiding line-connectivity constraints, checking for saturated subgraphs, enumerating subcatalogs, and alternative tree traversal strategies. Some theory, implementation details, and examples are provided for each enhancement.

*Ph.D candidate in Systems and Entrepreneurial Engineering, Department of Industrial and Enterprise Systems Engineering, University of Illinois at Urbana-Champaign, herber1@illinois.edu

†Assistant Professor, Department of Industrial and Enterprise Systems Engineering, University of Illinois at Urbana-Champaign, jtalliso@illinois.edu

Contents

List of Figures

List of Tables

List of Algorithms

1 Overview

In Ref. [1], a tree search algorithm was developed to generate a set of colored graphs covering the graph structure space defined by (C, R, P) and various additional network structure constraints. This algorithm is shown in Alg. 1 with some minor changes to the variable names and the feasible edge improvement directly included. Please refer to the original reference for additional theory and notation details. In this report, a number of enhancements are proposed to either more efficiently cover the same graph structure space or allow additional network structure constraints to be defined.

Algorithm 1: Original tree search algorithm.

Input : V – vector of remaining ports for each component replicate
E – vector of edges in sequential pairs, initially empty
A – expanded potential adjacency matrix
cVf – cumulative sum of the original V plus 1
SavedGraphs – set of graphs, initially empty
Output: SavedGraphs – set of graphs

```
1  iL ← find(V, first)                                  /* find first nonzero entry */
2  L ← cVf(iL) − V(iL)                                  /* left port */
3  V(iL) ← V(iL) − 1                                    /* remove port */
4  Vallow ← V ∘ A(iL,:)                                 /* zero infeasible edges */
5  I ← find(Vallow)                                     /* find nonzero entries */
6  for iR ← I do                                        /* loop through all nonzero entries */
7      R ← cVf(iR) − V(iR)                              /* right port */
8      E2 ← [E, L, R]                                   /* combine left, right ports for an edge */
9      V2 ← V                                           /* local remaining ports vector */
10     V2(iR) ← V2(iR) − 1                              /* remove port (local copy) */
11     A2 ← A                                           /* local expanded potential adjacency matrix */
12     if any element of V2 is nonzero then             /* recursive call if any remaining vertices */
13         SavedGraphs ← Algorithm 1 (tree) with V2, E2, A2, cVf, SavedGraphs
14     else
15         SavedGraphs{end + 1} ← E2                    /* save missorted perfect matching */
16     end
17 end
```

Each one of the enhancements will be discussed in the following format. First, the theory behind the enhancement will be presented with a focus on showing the desired graph structure space is still covered. Second, the implementation of the enhancement will be discussed with pseudocode. Finally, some examples are provided comparing the original algorithm to the enhancement. Both visualizations or other aides and computational tests are provided[1]. *En* is an abbreviation for enhancements included, *Orig* is an abbreviation for original algorithm (see Alg. 1), *NSC* is an abbreviation for network structure constraint (see Ref.[1]), and *12T* is an abbreviation for 12 threads.

There are seven enhancements proposed in this report: replicate ordering, avoiding loops, avoiding multi-edges, avoiding line-connectivity constraints, checking for saturated subgraphs, enumerating subcatalogs, and alternative tree traversal strategies. The final section discusses the effect of the enhancements on the case studies in Ref. [1].

[1] A tests were performed on a personal computer with an i7-6800K at 3.8 GHz (up to 12 threads available), 32 GB DDR4 3200 MHz RAM, WINDOWS 10 64-bit, and MATLAB 2017a.

2 Replicate Ordering

This enhancement is based on replicate ordering and is similar to the port-ordering modification that the original tree algorithm is based on [1]. Now, we eliminate some component-type isomorphisms during the graph generation process.

2.1 Theory

Consider a single component type and its set of N replicates. Now, consider a single replicate numbered n and $n \neq 1$. During an iteration of the tree algorithm (Alg. 1), a single edge is added. If replicate $n-1$ still has no ports connected, then adding this edge to n will produce a graph isomorphic to the graph created when adding the edge to $n-1$. This claim is based on the component-type isomorphism issue where we have two identical replicates currently with no edges. Therefore, we only need to allow a connection to $n-1$ and not to n, and the tree algorithm will continue generating the desired graph structure space. When $n = 1$, an edge will always be allowed since there is no other replicate to compare against.

This enhancement is general since it does not rely on any NSCs being present.

2.2 Implementation

Algorithm 2: Limit potential connections based on replicate ordering.

Input : V – vector of remaining ports for each component replicate
Vf – initial vector of ports available for each component replicate
iInitRep – indices of the initial replicate for each component type
Output: Vordering – binary vector where 1 indicates an edge is possible, 0 if it is not

1 Vordering $\leftarrow$ `circshift`(V, [1]) $\neq$ Vf `/* check if left neighbor has a connection */`
2 Vordering(iInitRep) $\leftarrow$ 1 `/* initial replicates are always 1 */`

Algorithm 2 is the pseudocode for the implementation of this enhancement. The implementation is centered around the creation of a binary vector (Vordering) with length equal to the total number of components where unity indicates a connection is allowed and zero indicates it is not. This enhancement is implemented efficiently with the `circshift`(v, k) function. This function circularly shifts the elements in array v by k positions [2]. Based on the discussion above, we want to determine if $n-1$ has been connected to any other vertex. Using the initial vector of ports available for each component replicate (Vf), we `circshift` the vector of remaining ports for each component replicate (V) by one position to the right and compare these vectors (see line 1). A pair that is not equal indicates $n-1$ has at least one edge, so we now need to allow connections to n.

The procedure above will produce incorrect results for the initial replicates. However, we have that these initial replicates should never have their connection disallowed so we simply ensure that their index in Vordering is always unity (see line 2). The indices of the initial replicate for each component type (iInitRep) is calculated before the tree algorithm is called since it does not change throughout the graph generation procedure.

This enhancement is inserted between lines 3 and 4 of Alg. 1 and line 4 is changed to:

$$\text{Vallow} = \text{V} \circ \text{A(iL,:)} \circ \text{Vordering} \tag{1}$$

Now, Vordering can disallow connections in the same way as A(iL,:).

2.3 Examples

2.3.1 Example 1

The base three-tuple and NSCs for this example are specified as:

$$C = \{X\}, \quad R = [4], \quad P = [2], \quad \text{no additional NSCs} \tag{2}$$

The second and third inputs for Algorithm 2 are:

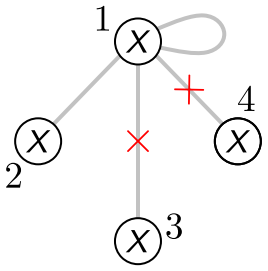
$$\mathsf{Vf} = [2\ 2\ 2\ 2], \quad \mathsf{iInitRep} = [1] \tag{3}$$

We will consider two different V, one at the initial iteration of the tree algorithm and one at some intermediate iteration. Figure 1a goes through the operations in Algorithm 2 for the different V. The example iterations are also visualized in Figs. 1b and 1c.

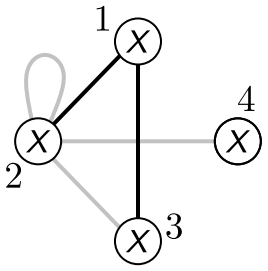
Table 1 compares the original algorithm with the enhancement for this example. There is a reduction in candidate graphs generated while the number of unique graphs remains the same.

	Initial	Intermediate
V	[1 2 2 2]	[0 0 1 2]
$\texttt{circshift}(\mathsf{V}[0\ 1])$	[2 1 2 2]	[2 0 0 1]
Vf	[2 2 2 2]	[2 2 2 2]
$\texttt{circshift}(\mathsf{V}[0\ 1]) \neq \mathsf{Vf}$	[0 1 0 0]	[0 1 1 1]
$\mathsf{Vordering}(\mathsf{iInitRep}) \leftarrow 1$	[1 1 0 0]	[1 1 1 1]
Removed/total branches	2/4	0/2

(a) Algorithm operations.



(b) Initial iteration.



(c) Intermediate iteration.

Figure 1: Example 1 for Algorithm 2 (replicate ordering).

	Orig	En	Orig/En
Candidate Graphs	26	8	3.25
Unique Graphs	5	5	1
Generation Time (s)	0.0052	0.0033	1.58
Total Time (s)	0.0095	0.0060	1.58

Table 1: Comparison (replicate ordering, Example 1).

2.3.2 Example 2

The base three-tuple and NSCs for this example are specified as:

$$C = \{W, X, Y, Z\}, \quad R = [3\ 4\ 2\ 1], \quad P = [1\ 2\ 2\ 3], \quad \text{no additional NSCs} \tag{4}$$

The second and third inputs for Algorithm 2 are:

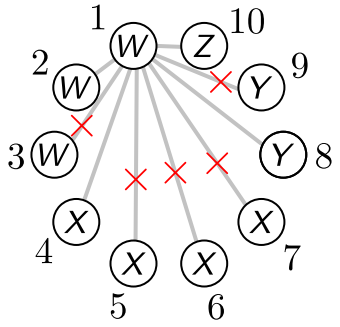
$$\mathsf{Vf} = [1\ 1\ 1\ 2\ 2\ 2\ 2\ 2\ 2\ 3], \quad \mathsf{iInitRep} = [1\ 4\ 8\ 10] \tag{5}$$

We will consider two different V, one at the initial iteration of the tree algorithm and one at some intermediate iteration. Figure 2a goes through the operations in Algorithm 2 for the different V. The example iterations are also visualized in Figs. 2b and 2c.

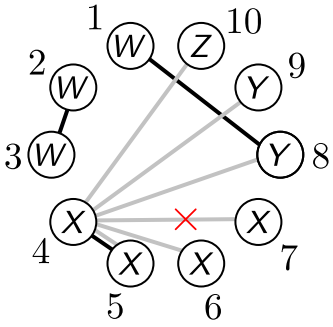
Table 2 compares the original algorithm with the enhancement for this example. There is a reduction in candidate graphs generated while the number of unique graphs remains the same.

	Initial	Intermediate
V	[0 1 1 2 2 2 2 2 2 3]	[0 0 0 0 1 2 2 1 2 3]
$\texttt{circshift}(\mathsf{V}[0\ 1])$	[3 0 1 1 2 2 2 2 2 2]	[3 0 0 0 0 1 2 2 1 2]
Vf	[1 1 1 2 2 2 2 2 2 3]	[1 1 1 2 2 2 2 2 2 3]
$\texttt{circshift}(\mathsf{V}[0\ 1]) \neq \mathsf{Vf}$	[1 1 0 1 0 0 0 0 0 1]	[1 1 1 1 1 1 0 0 1 1]
$\mathsf{Vordering}(\mathsf{iInitRep}) \leftarrow 1$	[1 1 0 1 0 0 0 1 0 1]	[1 1 1 1 1 1 0 1 1 1]
Removed/total branches	5/9	1/6

(a) Algorithm operations.



(b) Initial iteration.



(c) Intermediate iteration.

Figure 2: Example 2 for Algorithm 2 (replicate ordering).

	Orig	En	Orig/En
Candidate Graphs	456766	14359	31.8
Unique Graphs	1657	1657	1
Generation Time (s)	16.26	0.69	23.6
Total Time (s)	3577	145	24.7

Table 2: Comparison (replicate ordering, Example 2).

3 Avoiding Loops

Under specific NSCs, we can exclude loops during the graph generation process. A loop is an edge that connects a vertex to itself [3, p. 25].

3.1 Theory

Consider a component type that is both mandatory (S_3) and each edge is required to be unique (S_6). Then a feasible graph cannot have any loops for this component type since the number of edges would not be unique. We cannot make the same assumption if a component type needs to have each edge be unique but is not a mandatory component. Consider a two-port, nonmandatory component type with one replicate. Now, if we want a connected graph with all components except this specific two-port component, then a loop is required since each port must be filled. Therefore, this enhancement can be implemented with NAND logic where only a mandatory component type with each connection required to be unique is excluded from having loops.

This is not a general enhancement since it requires specific NSCs.

3.2 Implementation

Algorithm 3: Limit potential connections based on loops.

Input : A – expanded potential adjacency matrix
M – vector indicating if a component replicate is mandatory
U – vector indicating if a component replicate requires unique connections
Output: A – expanded potential adjacency matrix

```
1 if any(M ∧ U) then                          /* some loops should be excluded */
2     N ← length(M)                           /* total number of component replicates */
3     iDiag ← [1 : N + 1 : N^2]               /* indices for the diagonal elements */
4     A(iDiag) ← !(M ∧ U)                     /* assign NAND between M and U to the diagonal */
5 end
```

Algorithm 3 is the pseudocode for the implementation of this enhancement. The implementation is centered around modifying the expanded potential adjacency matrix (A) before the graph generation algorithm is called. The total number of components is found and then the linear index values for the diagonal of expanded potential adjacency matrix are computed. Finally, in line 4, NAND logical operator between M and U is assigned to the diagonal of A.

3.3 Examples

3.3.1 Example 1

The base three-tuple and NSCs for this example are specified as:

$$C = \{X, Y\}, \quad R = [2\ 2], \quad P = [2\ 2], \quad S_3 \text{ with } M = [0\ 1], \quad S_6 \text{ with } U = [1\ 1] \tag{6}$$

We will consider an all unity A but this could be any expanded potential adjacency matrix. The second and third inputs to Alg. 3 are:

$$\mathsf{M} = [0\ 0\ 1\ 1], \quad \mathsf{U} = [1\ 1\ 1\ 1], \quad !(\mathsf{M} \wedge \mathsf{U}) = [1\ 1\ 0\ 0] \tag{7}$$

Now, Alg. 3 modifies the expanded potential adjacency matrix as:

$$\mathsf{A} = \begin{array}{c} \\ X \\ X \\ Y \\ Y \end{array} \begin{array}{c} \begin{array}{cccc} X & X & Y & Y \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array} \xrightarrow{\mathsf{A}(\mathsf{iDiag}) \leftarrow !(\mathsf{M} \wedge \mathsf{U})} \mathsf{A} = \begin{array}{c} \\ X \\ X \\ Y \\ Y \end{array} \begin{array}{c} \begin{array}{cccc} X & X & Y & Y \end{array} \\ \begin{bmatrix} \mathbf{1} & 1 & 1 & 1 \\ 1 & \mathbf{1} & 1 & 1 \\ 1 & 1 & \mathbf{0} & 1 \\ 1 & 1 & 1 & \mathbf{0} \end{bmatrix} \end{array} \tag{8}$$

where Y is mandatory and unique connections are required so corresponding diagonal entries in A were zeroed. Table 3 compares the original algorithm with the enhancement for this example. There is a reduction in candidate graphs generated while the number of unique graphs remains the same.

	Orig	En	Orig/En
Candidate Graphs	26	16	1.63
Unique Graphs	3	3	1
Generation Time (s)	0.0024	0.0022	1.09
Total Time (s)	0.0086	0.0070	1.23

Table 3: Comparison (loops, Example 1).

3.3.2 Example 2

The base three-tuple and NSCs for this example are specified as:

$$C = \{X, Y\}, \quad R = [2\ 2], \quad P = [2\ 2], \quad S_3 \text{ with } M = [1\ 1], \quad S_6 \text{ with } U = [1\ 1] \tag{9}$$

We will consider an all unity A but this could be any expanded potential adjacency matrix. The second input to Alg. 3 is:

$$\mathsf{M} = [1\ 1\ 1\ 1], \quad \mathsf{U} = [1\ 1\ 1\ 1], \quad !(\mathsf{M} \wedge \mathsf{U}) = [0\ 0\ 0\ 0] \tag{10}$$

Now, Alg. 3 modifies the expanded potential adjacency matrix as:

$$\mathsf{A} = \begin{array}{c} \\ X \\ X \\ Y \\ Y \end{array} \begin{array}{c} \begin{array}{cccc} X & X & Y & Y \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array} \xrightarrow{\mathsf{A}(\mathsf{iDiag}) \leftarrow !(\mathsf{M} \wedge \mathsf{U})} \mathsf{A} = \begin{array}{c} \\ X \\ X \\ Y \\ Y \end{array} \begin{array}{c} \begin{array}{cccc} X & X & Y & Y \end{array} \\ \begin{bmatrix} \mathbf{0} & 1 & 1 & 1 \\ 1 & \mathbf{0} & 1 & 1 \\ 1 & 1 & \mathbf{0} & 1 \\ 1 & 1 & 1 & \mathbf{0} \end{bmatrix} \end{array} \tag{11}$$

Since both component types are mandatory and unique connections are required, the entire diagonal is zeroed. Table 4 compares the original algorithm with the enhancement for this example. There is a reduction in candidate graphs generated while the number of unique graphs remains the same.

	Orig	En	Orig/En
Candidate Graphs	26	11	2.36
Unique Graphs	2	2	1
Generation Time (s)	0.0024	0.0022	1.09
Total Time (s)	0.0086	0.0060	1.43

Table 4: Comparison (loops, Example 2).

4 Avoiding Multi-Edges

Under specific NSCs, we can exclude multi-edges during the graph generation process. A multi-edge is two or more edges that are incident to the same two vertices [3, p. 25].

4.1 Theory

Consider when each edge is required to be unique (S_6). Due to the sequential nature of the tree algorithm, a single edge must be added between two components before a second edge is added; thus, creating a multi-edge. Therefore, when the first edge is added between two components, we can utilize the expanded potential adjacency matrix to disallow any further connections between the components. Since a feasible graph would not have any multi-edges when each edge is required to be unique, the tree algorithm with this enhancement will continue generating the desired graph structure space.

This enhancement should not be applied on loops since loops are occasionally needed to remove components (see Sec. 3 for the handling of loops). Also, this is not a general enhancement since it requires specific NSCs.

4.2 Implementation

Algorithm 4: Limit potential connections based on multi-edges.

Input : A – expanded potential adjacency matrix
U – vector indicating if a component replicate requires unique connections
iR – component index for right port
iL – component index for left port
Output: A – expanded potential adjacency matrix

```
1 if U(iL) ∨ U(iR) then          /* either component requires unique connections */
2 |   if iR ≠ iL then                                  /* don't do for self loops */
3 |   |   A(iR, iL) ← 0                                 /* limit this connection */
4 |   |   A(iL, iR) ← 0                                 /* limit this connection */
5 |   end
6 end
```

Algorithm 4 is the pseudocode for the implementation of this enhancement. The implementation is centered around modifying the expanded potential adjacency matrix (A) when a new edge is created. This enhancement is only called if either component replicate in the edge requires unique connections. Additionally, this enhancement is only called if the edge is not a loop (see Sec. 3 for the handling of loops). If both of these conditions are met, then the corresponding entries in the expanded potential adjacency matrix are zeroed in lines 3 and 4.

This enhancement is inserted between lines 11 and 12 of Alg. 1 using the local copy A2.

4.3 Example

The base three-tuple and NSCs for this example are specified as:

$$C = \{W, X, Y, Z\}, \quad R = [1\ 1\ 1\ 1], \quad P = [3\ 3\ 3\ 3], \quad S_6 \text{ with } U = [1\ 1\ 1\ 1] \quad (12)$$

Consider one path during the graph generation process when the reduced potential adjacency matrix is initially all ones:

$$\begin{array}{c} \begin{array}{c} \\ W \\ X \\ Y \\ Z \end{array} \begin{array}{c} \begin{array}{cccc} W & X & Y & Z \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array} \rightarrow \begin{array}{c} \begin{array}{cccc} W & X & Y & Z \end{array} \\ \begin{bmatrix} 1 & \mathbf{0} & 1 & 1 \\ \mathbf{0} & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array} \rightarrow \begin{array}{c} \begin{array}{cccc} W & X & Y & Z \end{array} \\ \begin{bmatrix} 1 & 0 & \mathbf{0} & 1 \\ 0 & 1 & 1 & 1 \\ \mathbf{0} & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array} \rightarrow \\ \overbrace{\qquad\qquad}^{\text{Iter. 0, } \mathsf{V} = [3\ 3\ 3\ 3]} \qquad \overbrace{\qquad\qquad}^{\text{Iter. 1, } \mathsf{V} = [\bar{2}\ \bar{2}\ 3\ 3]} \qquad \overbrace{\qquad\qquad}^{\text{Iter. 2, } \mathsf{V} = [\bar{1}\ \not{2}\ \bar{2}\ 3]} \end{array}$$

$$\begin{array}{c} \rightarrow \begin{array}{c} \\ W \\ X \\ Y \\ Z \end{array} \begin{array}{c} \begin{array}{cccc} W & X & Y & Z \end{array} \\ \begin{bmatrix} 1 & 0 & 0 & \mathbf{0} \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ \mathbf{0} & 1 & 1 & 1 \end{bmatrix} \end{array} \rightarrow \begin{array}{c} \begin{array}{cccc} W & X & Y & Z \end{array} \\ \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & \mathbf{0} & 1 \\ 0 & \mathbf{0} & 1 & 1 \\ 0 & 1 & 1 & 1 \end{bmatrix} \end{array} \rightarrow \begin{array}{c} \begin{array}{cccc} W & X & Y & Z \end{array} \\ \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & \mathbf{0} \\ 0 & 0 & 1 & 1 \\ 0 & \mathbf{0} & 1 & 1 \end{bmatrix} \end{array} \\ \overbrace{\qquad\qquad}^{\text{Iter. 3, } \mathsf{V} = [\bar{0}\ \not{2}\ \not{2}\ \bar{2}]} \qquad \overbrace{\qquad\qquad}^{\text{Iter. 4, } \mathsf{V} = [0\ \bar{1}\ \bar{1}\ 2]} \qquad \overbrace{\qquad\qquad}^{\text{Iter. 5, } \mathsf{V} = [0\ \bar{0}\ \not{1}\ \bar{1}]} \end{array}$$

where the matrix above represents A, $\bar{\square}$ indicates this component was selected for an edge, and $\not{\square}$ indicates the connection was disallowed. Each iteration added a pair of zeros to the potential adjacency matrix.

Table 5 compares the original algorithm with the enhancement for this example. There is a reduction in candidate graphs generated while the number of unique graphs remains the same.

	Orig	En	Orig/En
Candidate Graphs	211	46	4.59
Unique Graphs	1	1	1
Generation Time (s)	0.0086	0.0037	2.32
Total Time (s)	0.015	0.0070	2.14

Table 5: Comparison (multi-edge).

5 Avoiding Line-Connectivity Constraints

On line 4 of Alg. 1, we utilize the expanded potential adjacency matrix (A) to disallow connections between components. This is also phrased as every graph must have edges between vertices that are feasible. These are termed vertex-connectivity constraints, denoted S_7. A similar type of NSC can be included between the lines of the graph, termed line-connectivity constraints.

5.1 Theory

The line graph of a graph G is the graph with the edges of G as its vertices, and where two edges of G are adjacent in the line graph if and only if they are incident in G [4, p. 10]. Consider the graph in Fig. 3a and its corresponding line graph in Fig. 3b with three component types. If component type 1 is connected to component type 2, we can specify if a connection between component types 2 and 3 is allowed. This is equivalent to specifying if line type $(1, 2)$ can be connected to line type $(2, 3)$.

This enhancement is a new type of NSC and is designated S_8.

(a) G. **(b)** Line graph of G.

Figure 3: Illustration of a line-connectivity constraint.

5.2 Implementation

Since this enhancement is a new type of NSC, it is specified before the graph generation procedure. For each line-connectivity constraint, a triple of integers is supplied defining the component types in Fig. 3a. They are supplied in increasing order, i.e., $[\#1, \#2, \#3]$. Therefore, each triple is interpreted as: if #1 and #2 are connected, don't ever connect #2 to #3. These triples help construct the reduced 3-D array with line-connectivity constraint information, B. They are indexed in reverse order to facilitate extracting column vectors, i.e., $B(\#3, \#2, \#1) = 0$. Given a set of triples, a function creates the expanded $N \times N \times N$ matrix where N is the total number of component replicates where a zero indicates a connection is not allowed and one indicates it is allowed.

Algorithm 5: Limit potential connections based on line-connectivity constraints.

Input : A – expanded potential adjacency matrix
iR – component index for right port
iL – component index for left port
B – 3-D array with line-connectivity constraint information
Output: A – expanded potential adjacency matrix

1 **if** *there are any line-connectivity constraints* **then**
2 $\quad \mathsf{A}(:, \mathsf{iR}) \leftarrow \mathsf{A}(:, \mathsf{iR}) \circ \mathsf{B}(:, \mathsf{iR}, \mathsf{iL})$ `/* potentially limit connections */`
3 $\quad \mathsf{A}(:, \mathsf{iL}) \leftarrow \mathsf{A}(:, \mathsf{iL}) \circ \mathsf{B}(:, \mathsf{iL}, \mathsf{iR})$ `/* potentially limit connections */`
4 $\quad \mathsf{A}([\mathsf{iR}, \mathsf{iL}], :) \leftarrow \mathsf{A}(:, [\mathsf{iR}, \mathsf{iL}])^{\mathsf{T}}$ `/* make symmetric */`
5 **end**

Algorithm 5 is the pseudocode for the implementation of this enhancement. This enhancement is only called if there are any line-connectivity constraints, S_8. First

using the component index of the right port, we extract a vector from the 3-D array with line-connectivity constraint information (B) when #1 = iL and #2 = iR. This vector is multiplied element-wise with the correct row of the expanded potential adjacency matrix (A), potentially limiting connections. This step then performed again switching the roles of component indices, i.e., #1 = iR and #2 = iL. Finally, the changes to A are applied to the symmetric location, ensuring A remains symmetric.

This enhancement is inserted between lines 11 and 12 of Alg. 1 using the local copy A2.

5.3 Examples

5.3.1 Example 1

The base three-tuple and NSCs for this example are specified as:

$$C = \{X, Y, Z\}, \quad R = [1\ 2\ 2], \quad P = [2\ 2\ 2], \quad S_8(1) = [1, 2, 3] \tag{13}$$

The reduced (B) and expanded (B) 3-D arrays with line-connectivity constraint information are:

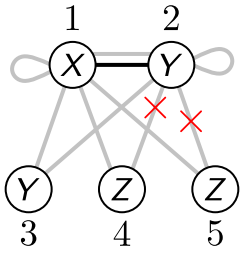
$$B(:,:,1) = \begin{array}{c} \\ X \\ Y \\ Z \end{array} \begin{array}{c} \begin{array}{ccc} X & Y & Z \end{array} \\ \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix} \end{array}, \quad \mathsf{B}(:,:,1) = \begin{array}{c} \\ X \\ Y \\ Y \\ Z \\ Z \end{array} \begin{array}{c} \begin{array}{ccccc} X & Y & Y & Z & Z \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 \end{bmatrix} \end{array} \tag{14}$$

Figure 4a goes through the operations in Algorithm 5 for a certain line type. The limiting of potential edges is visualized in Fig. 4b.

Table 6 compares the original algorithm with the enhancement for this example. There is a reduction in candidate graphs generated while the number of unique graphs remains the same.

	Line Type 1
iL	1
iR	2
B(:, iR, iL)	$[1\ 1\ 1\ 0\ 0]^\mathsf{T}$
B(:, iL, iR)	$[1\ 1\ 1\ 1\ 1]^\mathsf{T}$

(a) Algorithm operations.



(b) Line Type 1.

Figure 4: Example 1 for Algorithm 5 (line-connectivity constraints).

	Orig	En	Orig/En
Candidate Graphs	146	98	1.49
Unique Graphs	26	26	1
Generation Time (s)	0.004	0.005	0.80
Total Time (s)	0.036	0.034	1.06

Table 6: Comparison (line-constraints, Example 1).

5.3.2 Example 2

The base three-tuple and NSCs for this example are specified as:

$$\begin{aligned} C = \{\mathsf{X}, \mathsf{Y}, \mathsf{Z}\}, \quad R = [2\ 2\ 3], \quad P = [2\ 2\ 2] \\ S_8(1) = [1, 2, 2], \quad S_8(2) = [2, 1, 2], \quad S_8(3) = [3, 3, 3] \end{aligned} \tag{15}$$

The reduced 3-D array with line-connectivity constraint information (B) is:

$$\begin{aligned} B(:,:,1) = \begin{array}{c} \\ \mathsf{X} \\ \mathsf{Y} \\ \mathsf{Z} \end{array} \begin{array}{c} \begin{array}{ccc} \mathsf{X} & \mathsf{Y} & \mathsf{Z} \end{array} \\ \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix} \end{array}, \quad B(:,:,2) = \begin{array}{c} \\ \mathsf{X} \\ \mathsf{Y} \\ \mathsf{Z} \end{array} \begin{array}{c} \begin{array}{ccc} \mathsf{X} & \mathsf{Y} & \mathsf{Z} \end{array} \\ \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \end{array} \\ B(:,:,3) = \begin{array}{c} \\ \mathsf{X} \\ \mathsf{Y} \\ \mathsf{Z} \end{array} \begin{array}{c} \begin{array}{ccc} \mathsf{X} & \mathsf{Y} & \mathsf{Z} \end{array} \\ \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 0 \end{bmatrix} \end{array} \end{aligned} \tag{16}$$

The expanded 3-D arrays with line-connectivity constraint information (B) for $S_8(3)$ are:

$$\mathsf{B}(:,:,5) = \mathsf{B}(:,:,6) = \mathsf{B}(:,:,7) = \begin{array}{c} \\ \mathsf{X} \\ \mathsf{X} \\ \mathsf{Y} \\ \mathsf{Y} \\ \mathsf{Z} \\ \mathsf{Z} \\ \mathsf{Z} \end{array} \begin{array}{c} \begin{array}{ccccccc} \mathsf{X} & \mathsf{X} & \mathsf{Y} & \mathsf{Y} & \mathsf{Z} & \mathsf{Z} & \mathsf{Z} \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 \end{bmatrix} \end{array} \tag{17}$$

Figure 5a goes through the operations in Algorithm 5 for two line types. The limiting of potential edges is visualized in Figs. 5b and 5c.

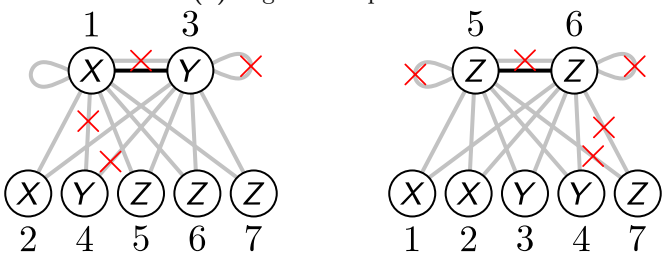
Table 7 compares the original algorithm with the enhancement for this example. There is a reduction in candidate graphs generated while the number of unique graphs remains the same.

	Orig	En	Orig/En
Candidate Graphs	8316	5120	1.62
Unique Graphs	119	119	1
Generation Time (s)	0.184	0.183	1.01
Total Time (s)	2.361	2.096	1.13

Table 7: Comparison (line-constraints, Example 2).

	Line Type 1	Line Type 2
iL	1	5
iR	3	6
B(:, iR, iL)	$[1\ 1\ 0\ 0\ 1\ 1\ 1]^\mathsf{T}$	$[1\ 1\ 1\ 1\ 0\ 0\ 0]^\mathsf{T}$
B(:, iL, iR)	$[1\ 1\ 0\ 0\ 1\ 1\ 1]^\mathsf{T}$	$[1\ 1\ 1\ 1\ 0\ 0\ 0]^\mathsf{T}$

(a) Algorithm operations.



(b) Line Type 1.

(c) Line Type 2.

Figure 5: Example 2 for Algorithm 5 (line-connectivity constraints).

6 Checking for Saturated Subgraphs

This enhancement is based on the work in Ref. [5] for enumerating molecules. In their work, all atoms are mandatory in the graph. Therefore, the detection of a saturated subgraph before all atoms have been connected indicates the candidate graph will be infeasible and can be discarded.

6.1 Theory

A subgraph of a graph G is another graph formed from a subset of the vertices and edges of G [3, p. 3]. A saturated subgraph is a subgraph with no empty ports and may contain multiple connected subgraphs [5]. Since a saturated subgraph has no empty ports, no components other than the components currently in this subgraph will be connected to this subgraph during further iterations of the graph generation procedure. If we determine that the current iteration of the tree algorithm has created a saturated subgraph, then there are three scenarios to consider.

First, if all mandatory components are contained in the saturated subgraph, then no additional iterations are needed. Since all components not connected to a mandatory component will be removed, all components not currently in the saturated subgraph will be removed. Therefore, the topology of the remaining components is negligible. Since the topology is negligible, we can assign an arbitrary topology to the remaining components, save the graph, and terminate the iteration.

The second scenario is if none of the mandatory components are in the saturated subgraph. This provides no additional information so we allow the current iteration to continue. The final scenario is when some, but not all, mandatory components are in the saturated subgraph. Since at least one pair of mandatory components will not be connected, this graph will be infeasible. Therefore, we can terminate this iteration without saving the graph.

This is not a general enhancement since it requires specific NSCs, namely at least one mandatory component (S_3).

6.2 Implementation

Algorithm 6 is the pseudocode for the implementation of this enhancement. This enhancement is only called if there are some mandatory components (S_3). First, the unsaturated components are found by checking the vector of remaining ports for each component replicate, V. A component is saturated if all ports are filled. To determine if the current graph is a saturated subgraph, we compare the remaining ports of the unsaturated components to the original number of ports available (see line 3).

If the current graph is indeed a saturated subgraph, then we compute the number of mandatory components not in the saturated subgraph, nUncon. If all mandatory components are in the saturated subgraph, then this graph is feasible. We assign an arbitrary ordering to the remaining components, save the graph, and terminate the iteration (see lines 6 to 13). If no mandatory components are in the saturated subgraph, then we allow the current iteration to continue (see line 14). Finally, if some but not all mandatory components are in the saturated subgraph, we stop the iteration since the graph is infeasible (see lines 15 and 16).

This enhancement is inserted between lines 11 and 12 of Alg. 1 since the current edge needs to be added but before the recursion call. With this enhancement, the `else` statement on line 14 of Alg. 1 will never be reached if there are any mandatory

Algorithm 6: Handle saturated subgraphs.

Input : V – vector of remaining ports for each component replicate
E – vector of edges in sequential pairs
Vf – initial vector of ports available for each component replicate
M – vector indicating if all replicates of the component type must be present
cVf – cumulative sum of the original V plus 1
SavedGraphs – set of graphs
Output: SavedGraphs – set of graphs

```
if there are any necessary components then
    iNonSat ← find(V)                                   /* find the nonsaturated components */
    if V(iNonSat) = Vf(iNonSat) then                    /* check for saturated subgraph */
        nUncon ← sum(M(iNonSat))                        /* # of mandatory comps not in sat subgraph */
        if nUncon = 0 then                              /* all mandatory components are in saturated subgraph */
            for j ← 1 to sum(V) do                      /* add remaining edges in arbitrary order */
                k ← find(V, 1)                          /* find first nonzero entry */
                LR ← cVf(k) − V(k)
                V(k) ← V(k) − 1                         /* remove port */
                E ← [E, LR]                             /* add port */
            end
            SavedGraphs{end + 1} ← E                    /* missorted perfect matching */
            continue                                    /* stop iteration, graph has been added */
        else if nUncon = sum(M) then                    /* no mandatory comps are in sat subgraph */
            /* continue with this iteration                                                       */
        else                                            /* some but not all mandatory components are in saturated subgraph */
            continue                                    /* stop iteration, this graph is infeasible */
        end
    end
end
```

components. The `if` condition is only untrue when a saturated subgraph is present (every component's port being filled).

6.3 Examples

6.3.1 Example 1

The base three-tuple and NSCs for this example are specified as:

$$C = \{X, Y\}, \quad R = [2\ 3], \quad P = [2\ 2], \quad S_3 \text{ with } M = [1\ 0] \tag{18}$$

Some of the other inputs are then:

$$\mathsf{Vf} = [2\ 2\ 2\ 2\ 2], \quad \mathsf{M} = [1\ 1\ 0\ 0\ 0] \tag{19}$$

We will consider two different V, one after an initial edge is added and one at some intermediate iteration. Figure 6a goes through the operations in Algorithm 6 for the different V. These example iterations are also visualized in Figs. 6b and 6c.

Table 8 compares the original algorithm with the enhancement for this example. There is a reduction in candidate graphs generated while the number of unique graphs remains the same.

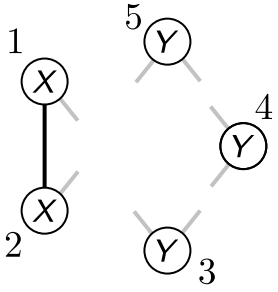
6.3.2 Example 2

The base three-tuple and NSCs for this example are specified as:

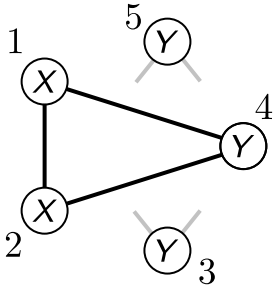
$$C = \{X\}, \quad R = [9], \quad P = [2], \quad S_3 \text{ with } M = [1] \tag{20}$$

Iteration	Initial	Intermediate
V	[1 1 2 2 2]	[0 0 2 0 2]
iNonSat ← `find`(V)	[1 2 3 4 5]	[3 5]
V(iNonSat)	[1 1 2 2 2]	[2 2]
Vf(iNonSat)	[2 2 2 2 2]	[2 2]
V(iNonSat) = Vf(iNonSat)	False	True
M(iNonSat)	–	[0 0]
nUncon ← `sum`(M(iNonSat))	–	0
Feasible	–	Yes

(a) Algorithm operations.



(b) Initial iteration.



(c) Intermediate iteration.

Figure 6: Example 1 for Algorithm 6 (saturated subgraphs).

	Orig	En	Orig/En
Candidate Graphs	146	91	1.60
Unique Graphs	6	6	1
Generation Time (s)	0.0056	0.0046	1.22
Total Time (s)	0.023	0.015	1.53

Table 8: Comparison (saturated subgraphs, Example 1).

Some of the other inputs are then:

$$\mathsf{Vf} = [2\ 2\ 2\ 2\ 2\ 2\ 2\ 2\ 2], \quad \mathsf{M} = [1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1] \tag{21}$$

We will consider two different V, one after an initial edge is added and one at some intermediate iteration. Figure 7a goes through the operations in Algorithm 6 for the different V. These example iterations are also visualized in Figs. 7b and 7c.

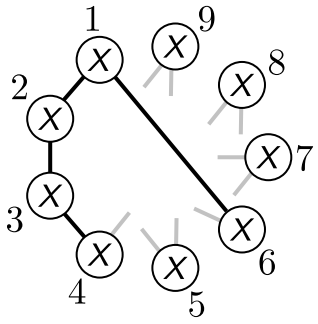
Table 9 compares the original algorithm with the enhancement for this example. There is a reduction in candidate graphs generated while the number of unique graphs remains the same.

	Orig	En	Orig/En
Candidate Graphs	852316	460872	1.85
Unique Graphs	1	1	1
Generation Time (s)	29.83	19.23	1.55
Total Time (s)	58.57	33.11	1.77

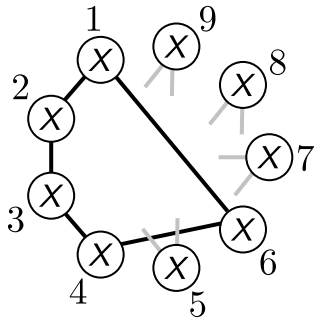
Table 9: Comparison (saturated subgraphs, Example 2).

	Intermediate 1	Intermediate 2
V	[0 0 0 1 2 1 2 2 2]	[0 0 0 0 2 0 2 2 2]
iNonSat $\leftarrow$ `find`(V)	[4 5 6 7 8 9]	[5 7 8 9]
V(iNonSat)	[1 2 1 2 2 2]	[2 2 2 2]
Vf(iNonSat)	[2 2 2 2 2 2]	[2 2 2 2]
V(iNonSat) = Vf(iNonSat)	False	True
M(iNonSat)	–	[1 1 1 1]
nUncon $\leftarrow$ `sum`(M(iNonSat))	–	4
Feasible	–	No

(a) Algorithm operations.



(b) Intermediate 1 iteration.



(c) Intermediate 2 iteration.

Figure 7: Example 2 for Algorithm 6 (saturated subgraphs).

7 Enumerating Subcatalogs

Leveraging some of the properties of the graph structure space and some NSCs, we can break the graph generation procedure into subtasks that more efficiently generate the same graph structure space.

7.1 Theory

A candidate architecture in an architecture design space described by (C, R, P) has the following properties [1]:

1. A set of component replicates bounded by (C, R, P). This set of component replicates is termed a subcatalog of (C, R, P).
2. Each port in $(V^P, \{\}, L^P)$, i.e., G^P without edges, is connected to another port (this implies an even number of ports).

The original tree algorithm in Alg. 1 generated all candidate architectures in this architecture design space since the set of PMs graphs of K_N contains all edge sets for K_{N-2}, where $N \geq 4$ [1]. Instead of relying on this property, an alternative would be an enumeration of all possible subcatalogs of (C, R, P). This property is no longer strictly needed since the edge sets of the PMs of K_N for each subcatalog is enough to generate all the desired graphs. However, this approach on its own provides no general improvements to the graph generation procedure.

If we require every graph to be a connected graph (S_1), we can enforce the following: A feasible graph for a specific subcatalog must have every component replicate connected (i.e., all the replicates are mandatory). This is due to the tree algorithm being utilized on every subcatalog. Enumerating subcatalogs only provides general improvements to the graph generation procedure if the property that all replicates are mandatory in each subcatalog is effectively utilized.

Two of the previously discussed enhancements utilize this property: 1) checking for saturated subgraphs in Sec. 6 and 2) avoiding loops in Sec. 3. A greater proportion of mandatory component types improves the effectiveness of these enhancements. Another benefit is isomorphism checks only need to be performed between graphs in their respective subcatalog. Since every component type is mandatory in the subcatalog, the colored label sets will be different between graphs in different subcatalogs. Therefore, graphs from different subcatalogs are definitely not isomorphic. This reduces the number of isomorphism checks and allows for further parallelization. Finally, the generation of graphs for each subcatalog can be performed in parallel. However, each subcatalog will take varying amounts of time to complete so the benefit will vary depending on the particular (C, R, P) and NSCs. The original tree algorithm does not leverage parallelization during the graph generation procedure.

This enhancement also allows for an improved representation of the number of replicates for each component type. Instead of the original vector R, where each entry was the maximum number of replicates for the specific component type, minimum and maximum values can be specified. The maximum values, denoted $R_{\max}$, is equivalent to the previous R. The minimum values, denoted $R_{\min}$, can naturally capture mandatory components and nonzero lower bounds. Every nonzero element of $R_{\min}$ indicates a mandatory component type. If $R_{\min}$ and $R_{\max}$ for a component type is 2 and 5, then there must be between 2 and 5 replicates in a feasible graph.

The set of subcatalogs contains all possible combinations of integers values for each component type bounded by $R_{\min}$ and $R_{\max}$. Therefore, the total number of subcatalogs is:

$$N_{\text{subcatalogs}} = \prod_{k=1}^{|R_{\max}|} \left[R_{\max}(k) - R_{\min}(k) + 1 \right] \tag{22}$$

Some of these subcatalogs may be invalid, e.g., the subcatalog has an odd number of ports or is empty.

7.2 Implementation

Algorithm 7: Generate set of unique, feasible graphs using subcatalogs.

Input : Rmin – vector indicating min number of replicates for each component type
Rmax – vector indicating max number of replicates for each component type
C – colored label set
P – ports vector
NSC – structure for the network structure constraints
Output: FinalGraphs – set of unique, feasible graphs

```
1  for k ← 1 to length(Rmax) do                          /* for each component type */
2  |  Rlist{k} ← Rmin(k) : 1 : Rmax(k)                   /* list of potential number of replicates */
3  end
4  Subcatalogs ← matrix form of the cell array from ndgrid(Rlist)   /* create subcatalogs */
5  Subcatalogs ← filter Subcatalogs (empty, odd port, custom filters)
6  Nsubcatalogs ← number of rows in Subcatalogs          /* number of subcatalogs */
7  nsc ← NSC                                             /* local NSC structure */
8  for k ← 1 to Nsubcatalogs do in parallel
9  |  r ← Subcatalogs(k, :)                             /* extract R vector for this subcatalog test */
10 |  I ← r ≠ 0                                         /* nonzero replicate locations */
11 |  c ← C(I)                                          /* extract colored labels */
12 |  r ← r(I)                                          /* extract replicates vector */
13 |  p ← P(I)                                          /* extract ports vector */
14 |  nsc.U ← NSC.U(I)                                  /* extract unique connections vector */
15 |  nsc.A ← NSC.A(I, :)                               /* extract reduced potential adjacency matrix */
16 |  nsc.A(:, I) ← NSC.A(:, I)                         /* symmetric */
17 |  nsc.B ← extract appropriate line-connectivity triples using NSC.B and I
18 |  nsc.M ← ones(size(r))                             /* all component types are mandatory */
19 |  Graphs{k} ← generate feasible graphs for this subcatalog using c, r, p, and nsc
20 end
21 for k ← 1 to Nsubcatalogs do in parallel            /* obtain unique graphs */
22 |  Graphs{k} ← determine set of unique graphs in Graphs{k}
23 end
24 FinalGraphs ← combine unique graphs from each subcatalog using Graphs
```

Algorithm 7 is the pseudocode for the implementation of this enhancement. This enhancement is only called if we require every graph to be a connected graph (S_1)

First, the potential number of replicates for each component type is stored in a cell array, Rlist. All subcatalogs are then generated with `ndgrid` using Rlist. This function creates a rectangular grid in N-D space [6]. Next, the subcatalogs are filtered for subcatalogs with an odd number of ports or are empty. Additional user-specified filters can all be applied here. Once all the filters have been applied, the number of subcatalogs is calculated. The final step before generating the graphs is to create a local copy of the network structure constraints since they are modified for each subcatalog.

Generating graphs for each subcatalog can now be performed in parallel (see line 8). Before a subcatalog is used, a number of items need to be updated to

properly define the subcatalog. We find the locations of the nonzero replicates on line 10. Then the colored labels, replicates vector, and ports vector are updated for this subcatalog, only including component types with at least one replicate. In addition, both the unique connections vector, reduced potential adjacency matrix, and line-connectivity triples need to be updated to include only the relevant constraints for this subcatalog. Some component types may not be present in a particular subcatalog, so any NSCs with these component types are not needed. Since all component types are mandatory in this approach, we set each component type as mandatory on line 18. Finally, we generate feasible graphs for this subcatalog using the same method used for a single catalog.

After the feasible graphs have been found, each subcatalog can be analyzed for the set of unique graphs. Again, this task can be performed in parallel as each subcatalog is independent. The final step is to combine all the unique graphs into a single set.

7.3 Examples

These examples are tested using this enhancement and the handling of saturated subgraphs in Sec. 6.

7.3.1 Example 1

The base three-tuple and NSCs for this example are specified as:

$$C = \{X, Y\}, \quad R_{\min} = [1\ 0], \quad R_{\max} = [1\ 8], \quad P = [2\ 2] \tag{23}$$

All 9 subcatalogs for this example are shown in Table 10.

Table 11 compares the original algorithm with the enhancement for this example. There is a reduction in feasible graphs generated while the number of unique graphs remains the same. The enhancement with 12 threads (12T) and 1 thread (1T) available is shown.

#	r	c	p	Feasible Graphs
1	[1]	$\{X\}$	[2]	1
2	[1 1]	$\{X, Y\}$	[2 2]	1
3	[1 2]	$\{X, Y\}$	[2 2]	1
4	[1 3]	$\{X, Y\}$	[2 2]	3
5	[1 4]	$\{X, Y\}$	[2 2]	12
6	[1 5]	$\{X, Y\}$	[2 2]	60
7	[1 6]	$\{X, Y\}$	[2 2]	360
8	[1 7]	$\{X, Y\}$	[2 2]	2520
9	[1 8]	$\{X, Y\}$	[2 2]	20160

Table 10: Subcatalogs for Example 1.

	Orig	En	En (12T)	Orig/En	Orig/En (12T)
Feasible Graphs	96940	23118	23118	4.19	4.19
Unique Graphs	9	9	9	1	1
Generation Time (s)	29.857	27.385	26.371	1.09	1.13
Total Time (s)	39.604	29.943	29.721	1.32	1.33

Table 11: Comparison (enumerating subcatalogs, Example 1).

7.3.2 Example 2

The base three-tuple and NSCs for this example are specified as:

$$C = \{W, X, Y, Z\}, \quad R_{\min} = [1\ 0\ 0\ 0], \quad R_{\max} = [1\ 2\ 3\ 3], \quad P = [1\ 1\ 2\ 3] \tag{24}$$

A select number of the 48 subcatalogs for this example are shown in Table 12.

Table 13 compares the original algorithm with the enhancement for this example. There is a reduction of feasible graphs generated while the number of unique graphs remains the same. The enhancement with 12 threads (12T) available is also shown.

#	r	c	p	Feasible Graphs
1	[1 0 0 0]	$\{W\}$	[1]	– (odd)
2	[1 1 0 0]	$\{W, X\}$	[1 1]	1
3	[1 2 0 0]	$\{W, X\}$	[1 1]	– (odd)
4	[1 0 1 0]	$\{W, Y\}$	[1 2]	– (odd)
5	[1 1 1 0]	$\{W, X, Y\}$	[1 1 2]	1
⋮	⋮	⋮	⋮	⋮
44	[1 1 2 3]	$\{W, X, Y, Z\}$	[1 1 2 3]	– (odd)
45	[1 2 2 3]	$\{W, X, Y, Z\}$	[1 1 2 3]	1548
46	[1 0 3 3]	$\{W, Y, Z\}$	[1 2 3]	1683
47	[1 1 3 3]	$\{W, X, Y, Z\}$	[1 1 2 3]	– (odd)
48	[1 2 3 3]	$\{W, X, Y, Z\}$	[1 1 2 3]	11844

Table 12: Select subcatalogs for Example 2.

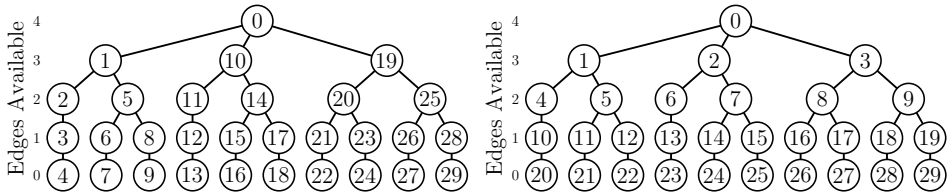
	Orig	En	En (12T)	Orig/En	Orig/En (12T)
Feasible Graphs	45015	16235	16235	2.77	2.77
Unique Graphs	489	489	489	1	1
Generation Time (s)	8.940	12.903	11.216	0.69	0.80
Total Time (s)	60.977	49.285	44.291	1.24	1.38

Table 13: Comparison (enumerating subcatalogs, Example 2).

8 Alternative Tree Traversal Strategies

Visualized in Ref. [1], the main algorithm in Alg. 1 for enumerating the graph structure space of interest is functionally equivalent to visiting all nodes in a tree, denoted τ. Here we will further characterize the tree structure and alternative strategies for traversing it. All the enhancements discussed in this appendix can be readily incorporated into the alternative tree traversal strategies discussed here.

A *tree* is an undirected graph in which any two vertices are connected by a unique path [7, p. 27]. A *rooted tree* is a tree in which one vertex has been designated the root [3, p. 13]. A *directed rooted tree* is a rooted tree where the edges are assigned a natural orientation, either away from or towards the root [7, p. 29]. Algorithm 1 traverses a *directed rooted tree*. Here the root of τ is G^P without edges (or a graph with all the ports, see Ref. [1]) and every vertex in τ represents some undirected labeled graph. The edges in τ are naturally directed away from the root because the algorithm produces new graphs in this direction by adding edges. Each directed edge (parent $\rightarrow$ child) in τ represents the addition of a single edge to the parent graph to create the child graph. The height of a vertex in a rooted tree is the length of the longest downward path to the vertex from the root. Then the height of τ is equal to half the number of ports (or the number of edges needed to create a perfect matching). Only the set of vertices in τ with maximal height comprise the graph structure space of interest.



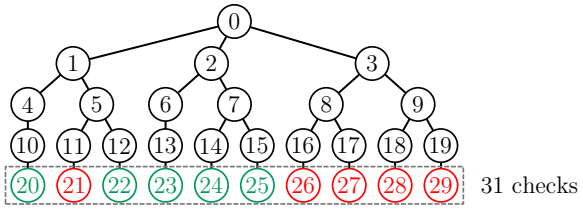
(a) Depth-first search. **(b)** Breadth-first search.

Figure 8: Two tree traversal strategies (numbers indicate order the vertices are visited).

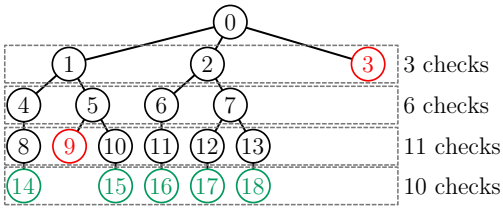
8.1 Depth-First vs. Breadth-First Search

Algorithm 1 was titled *Original Tree Search Algorithm*, but we can be more descriptive of the particular algorithm implementation. There are two basic methods of tree traversal (the process of visiting each vertex in a directed rooted tree): depth-first search (DFS) or breadth-first search (BFS) [8, 9]. The primary difference the two methods is the order in which the vertices are explored [8]. DFS explores a particular path in the tree to the maximum height possible before backtracking and continuing down an alternative, unexplored path [9]. This process is visualized in Fig. 8a. There are both stack-based and recursive implementations of the DFS [8, pp. 169–172]. From these definitions, Alg. 1 can be classified as a recursive DFS algorithm. While the current implementation works fairly well, the other tree traversal method, BFS, may be better suited for enumerating the graph structure space of interest.

A BFS method traverses the tree by visiting each vertex in a particular level first before moving to larger levels through the use of a queue [8, 9]. Levels are defined by sets of vertices with the same height. This process is visualized in Fig. 8b and



(a) Without level-order isomorphism checking.



(b) With level-order isomorphism checking.

Figure 9: Visualization of the impact of level-order isomorphism checking during the graph generation process for $(C, R, P) = (\{G, B\}, [1\ 2], [2\ 3])$.

note the difference between DFS. The potential advantage of a BFS implementation would be the ability to include isomorphism checking at each level. In the current implementation, this is not possible so (potentially) many intermediate graphs that are isomorphic to other intermediate graphs are enumerated. By identifying isomorphic intermediate graphs, we can remove these vertices (and their subtree) from the tree traversal process. Thus, there would be a reduction the number of graphs generated while covering the same graph structure space. However, there is a considerable computational cost associated with checking if a set of graphs is isomorphic (see Sec. 9), so there may be cases where the overall computational expense is larger with a BFS implementation with level-order isomorphism checking.

Consider the example in Fig. 9 comparing the BFS method with and without level-order isomorphism checking. The desired set of unique graphs (colored green) is covered by both approaches, but different trees are traversed. There is a reduction from a total of 29 generated graphs to 18, but the number of graph comparisons needed only decreases from 31 to 30. With the overhead associated with calling the isomorphism checking function, level-order isomorphism checking may actually increase computation time. This is even more likely with the enhancements included because some of the vertices would be removed faster through the enhancements rather than direct isomorphism checking. It is future work to both implement this enhancement and determine its impact on the overall computational expense.

8.2 Parallelized Tree Traversal

We can further leverage our knowledge of the tree structure by parallelizing the traversal process. There has been considerable work in parallelizing various graph algorithms [10, 11]. The enhancement in Sec. 7 for enumerating subcatalogs was a type of parallelized tree traversal, but it is more or less unpredictable in how it partitions the original tree (which was acceptable since it covers the same graph structure space). There are additional parallel traversal strategies that could be implemented in conjunction with the other enhancements such as the one below.

Consider a tree with N levels. We can proceed as normal on a single worker

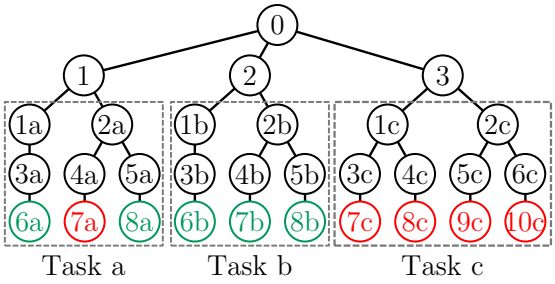


Figure 10: Parallelization example.

using the BFS method up to level $n \leq N$. At this point, the task of traversing the subtree of each vertex in level n is a parallelizable task (assuming no level-order isomorphism checking). Level-order isomorphism checking could still be utilized in each of the tasks, but the collected graphs from each of the tasks would still need to be checked for uniqueness. An example of this approach is shown in Fig. 10 with three tasks and assuming perfect parallelization of the generation task, a reduction from 29 to 13 effective algorithm calls (3 plus the maximum of the tasks).

A Project Code

The project code is available at Ref. [12]. The enhancements discussed in this report have been included in commit `7d8c14d`. Updated versions of the project code may contain modifications that change the implementation/presentation of the enhancements. Table 14 lists the enhancements and the functions that have been added/modified to implement the enhancement.

Enhancement	Functions
Replicate Ordering	`TreeEnumerateCreatev8`
Avoiding Loops	`ExpandPossibleAdj`
Avoiding Multi-Edges	`TreeEnumerateCreatev8`
Avoiding Line-Connectivity Constraints	`TreeEnumerateCreatev8`
	`CreateBMatrix`
Checking for Saturated Subgraphs	`TreeEnumerateCreatev8`
Enumerating Subcatalogs	`GenerateWithSubcatalogs`
	`UniqueUsefulGraphs`

Table 14: Changes in the codes to support the enhancements.

B Case Studies from Ref. [1]

In Tables. 15–17, the results from the case studies in Ref. [1] are compared with the enhancements in this report. All enhancements are present in the comparisons. No parallel computing was used in Case Study 1/2 and in Case Study 3, 12 threads were used for any parallel computing tasks. The PYTHON isomorphism checking method was used.

	Example 1			Example 2		
	Orig	En	Orig/En	Orig	En	Orig/En
Candidate Graphs	86	41	2.10	–	–	–
Feasible Graphs	77	39	1.97	23	11	2.09
Unique Graphs	16	16	1	5	5	1
Generation Time (s)	0.010	0.006	1.67	0.013	0.009	1.44
Total Time (s)	0.039	0.022	1.77	0.189	0.176	1.07

Table 15: Comparison (Case Study 1).

(a) Examples 1 and 2.

	Example 1			Example 2		
	Orig	En	Orig/En	Orig	En	Orig/En
Candidate Graphs	1119	633	1.77	–	–	–
Feasible Graphs	767	442	1.74	767	212	3.62
Unique Graphs	274	274	1	140	140	1
Generation Time (s)	0.082	0.051	1.61	0.111	0.107	1.04
Total Time (s)	1.514	1.211	1.25	0.352	0.322	1.09

(b) Examples 3 and 4.

	Example 3			Example 4		
	Orig	En	Orig/En	Orig	En	Orig/En
Feasible Graphs	31	22	1.41	34	25	1.36
Unique Graphs	12	12	1	14	14	1
Generation Time (s)	0.137	0.008	17.13	0.134	0.009	14.89
Total Time (s)	0.156	0.024	6.50	0.153	0.027	5.67

Table 16: Comparison (Case Study 2).

(a) Formulations changes with enhancements.

$$R_{\min} = [1\ 1\ 0\ 0\ 0\ 0\ 0\ 0], \quad R_{\max} = [1\ 1\ 2\ 2\ 2\ 1\ 2\ 2]$$
$$S_8(1) = [1, 7, 2], \quad S_8(2) = [2, 7, 1], \quad S_8(3) = [1, 8, 2], \quad S_8(4) = [2, 8, 1]$$

(b) Results.

	Orig	En	Orig/En
Feasible Graphs	1943862	48408	40.16
Unique Graphs	13727	13774	0.997
Generation Time (s)	10872.7	251.8	43.18
Total Time (s)	17903.2	688.1	26.02

Table 17: Comparison (Case Study 3).

References

[1] D. R. Herber, T. Guo, and J. T. Allison, "Enumeration of architectures with perfect matchings," *Journal of Mechanical Design*, vol. 139, no. 5, p. 051403, May 2017, doi: 10.1115/1.4036132

[2] The MathWorks. circshift. Accessed on Apr. 3, 2017. [Online]. Available: https://www.mathworks.com/help/matlab/ref/circshift.html

[3] R. Diestel, *Graph Theory*, electronic ed. Springer, 2000.

[4] C. Godsil and G. Royle, *Algebraic Graph Theory*. Springer, 2001.

[5] J.-L. Faulon, C. J. Churchwell, and D. P. Visco, "The signature molecular descriptor. 2. Enumerating molecules from their extended valence sequences," *Journal of Chemical Information and Computer Sciences*, vol. 43, no. 3, pp. 721–734, May 2003, doi: 10.1021/ci020346o

[6] The MathWorks. ndgrid. Accessed on Apr. 10, 2017. [Online]. Available: https://www.mathworks.com/help/matlab/ref/ndgrid.html

[7] E. L. Lawler, *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart and Winston, 1976.

[8] S. S. Skiena, *The Algorithm Design Manual*, 2nd ed. Springer, 2008, doi: 10.1007/978-1-84800-070-4

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. The MIT Press, 2009.

[10] M. J. Quinn and N. Deo, "Parallel graph algorithms," *Computing Surveys*, vol. 16, no. 3, pp. 319–348, Sep. 1984, doi: 10.1145/2514.2515

[11] E. Reghbati and D. G. Corneil, "Parallel computations in graph theory," *SIAM Journal on Computing*, vol. 7, no. 2, pp. 230–237, May 1978, doi: 10.1137/0207020

[12] D. R. Herber. PM architectures project. Accessed on Apr. 3, 2017. [Online]. Available: https://github.com/danielrherber/pm-architectures-project