Supplementary Data

Supplementary Figures

1) Datasets

Table 1 in the main text shows the run-time of the WaRSwapApp in three different TF-miRNA-gene regulatory networks. We selected two representative *Arabidopsis thaliana* networks (Network 1 and Network 3) from (Megraw et al., 2013), and a published *Drosophila* network (Network 2) from (Roy et al., 2010). 2500 different randomized background networks were generated and used to identify 3-node motifs. Runtimes for the PC version of WaRSwapApp were measured on a personal laptop with the following configuration:

- ***CPU-*** Intel(R) Core(TM) i5-3320M CPU @ 2.60GHz – 2 cores
- ***Memory-*** 4GB

The Cluster WaRSwapApp was deployed on a Sun Grid Engine (SGE) platform with 200 available cluster nodes when reporting run times in Table 1.

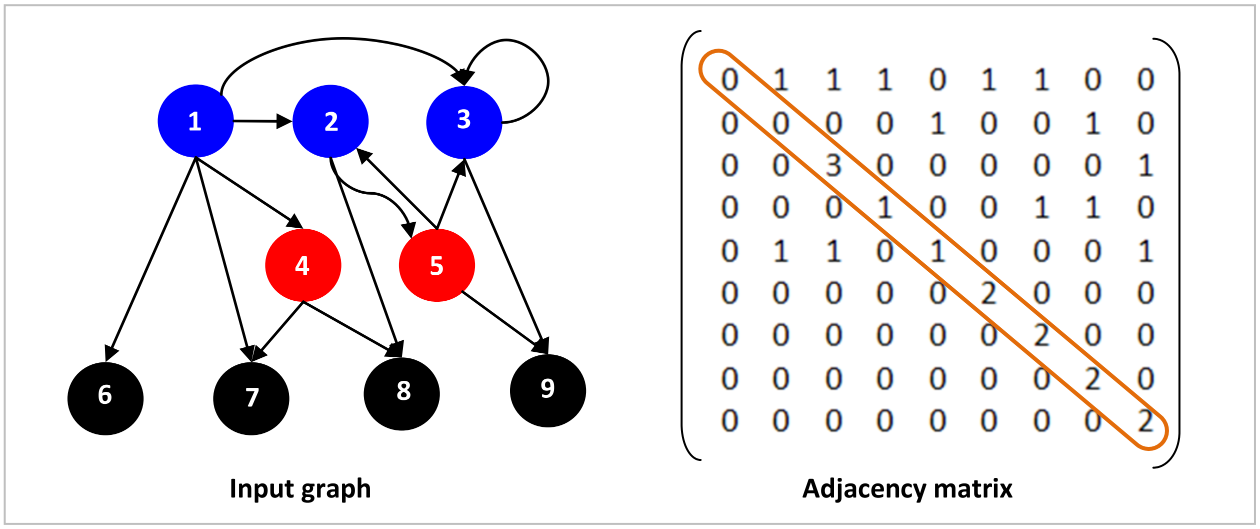
(The minimum system configuration for running the PC version of WaRSwapApp is recommended as follows: 2GHz CPU, 1 core, 4GB memory.)

2) Modifications to subgraph enumeration method (Subenum)

To enumerate subgraphs we built upon a recently published method Subenum (Shahrivari and Jalili, 2015), which performs subgraph enumeration without a need to call graph-isomorphism detection libraries. We selected this method for its advantages in run-time over previously available implementations of graph enumeration procedure. The published Subenum implementation is provided in java, and only supports input networks with a single node type. We modified the Subenum method to handle input graphs with three different node types (TFs, miRNAs, and non-TF genes). In the following sections we explain the modifications that we made to the published Subenum implementation.

A. Modifying adjacency matrix of graph

To handle input graphs with more than one node type, and also self-loops, we changed the adjacency matrix values such that matrix entry in row *i* and column *j* ($d_{i,j}$) can have integer values zero to three, in which the zero value represents a TF node, one represents a miRNA, two represents a non-TF gene, and three represents a self-loop. Supplementary Figure 1 shows an example of a multi-node type graph and its corresponding adjacency matrix.

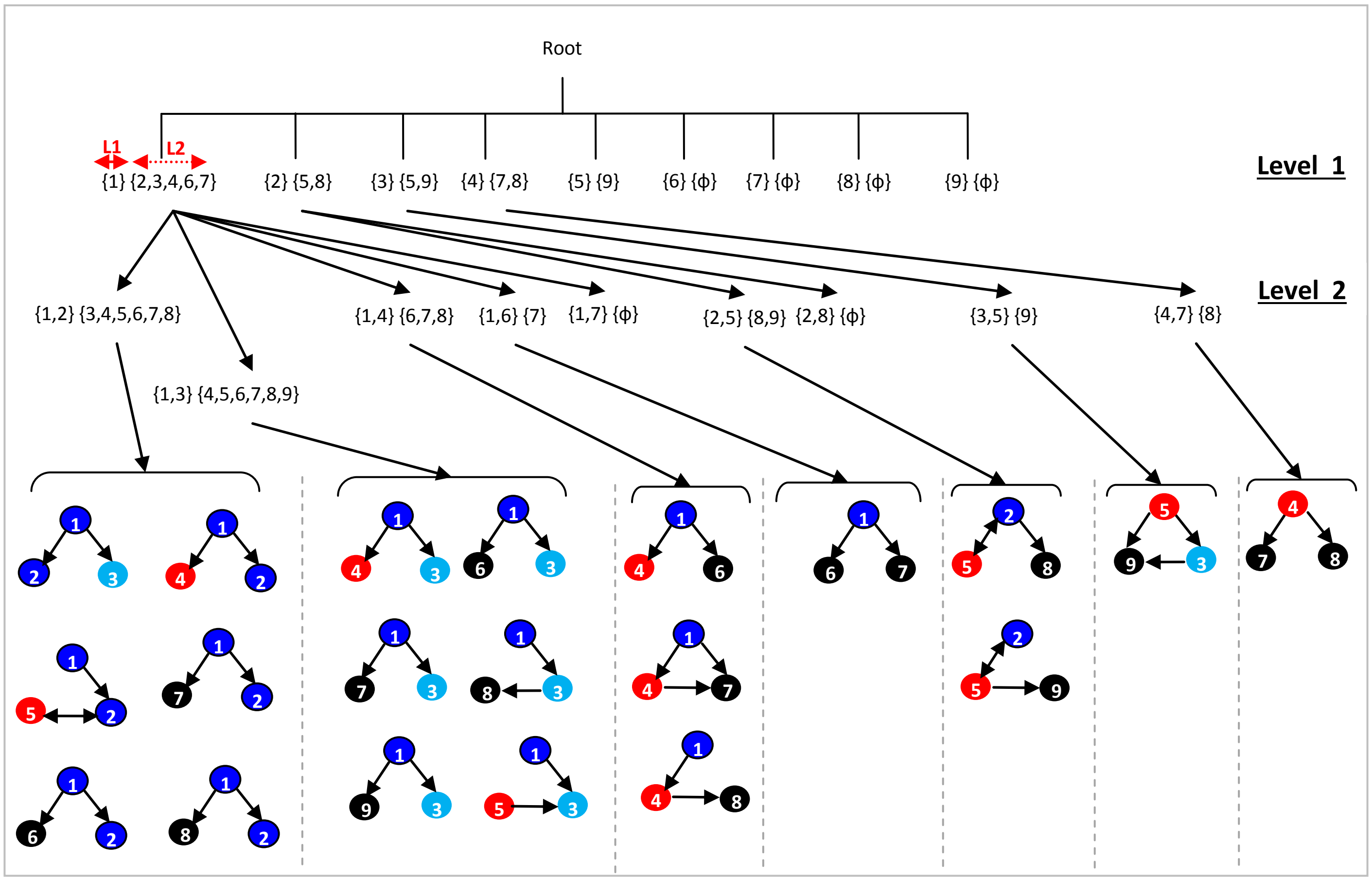


$$\begin{pmatrix} 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 3 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 \end{pmatrix}$$

Supplementary Figure 1: Example of a 3-layer graph and its corresponding adjacency matrix

B. Enumerating all subgraphs of size K

The Subenum method implements the ESU algorithm (Wernicke,S. 2005) for enumerating subgraphs of size K in an input graph. Supplementary Figure 2 shows a fully expanded search tree to enumerate 3-node subgraphs for the graph in Supplementary Figure 1. This part of the implementation is unmodified because the node types do not affect construction of the search tree. This procedure enumerates all subgraphs of size K as explained in the diagram below.

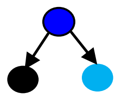
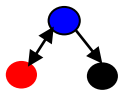
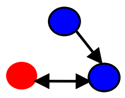
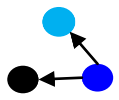


Supplementary Figure 2: Example of subgraph enumeration for multi-node graphs

This figure illustrates how a search tree is constructed using the ESU algorithm. Each node in the search tree keeps two lists of vertices: 1) partial subgraph vertices L1, and 2) the extension list L2 (red arrows in the figure highlight each list). The search tree expands to two levels, and the leaves of the tree represent 3-node subgraphs. Starting with vertex 1 at level 1 from the input graph, we add only those vertices to the extension list of vertex 1 (L2) such that vertex 1 has a direct edge to these vertices, and their label is larger than that of vertex 1. At the next level, the L1 and L2 extend in an analogous way to that described above. The algorithm finishes after expanding the tree for two levels.

C. Canonical labeling examples

A canonical labeling of a subgraph G is a subgraph G' which is isomorphic to G and represents the whole isomorphism class of G. Supplementary Figure 3 shows examples of canonical labeling for different subgraphs.

	Subgraph	Adj-matrix	Canonical label
(1)		$\begin{matrix} 0 & 1 & 1 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{matrix}$	300, 020, 110
(2)		$\begin{matrix} 0 & 1 & 1 \\ 1 & 1 & 0 \\ 0 & 0 & 2 \end{matrix}$	200, 011, 110
(3)		$\begin{matrix} 0 & 0 & 1 \\ 0 & 1 & 1 \\ 0 & 1 & 0 \end{matrix}$	110, 100, 010
(4)		$\begin{matrix} 3 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 2 \end{matrix}$	300, 020, 011

Supplementary Figure 3: Canonical labeling on example subgraphs

In this figure subgraphs 1 and 4 are isomorphic.

D. Canonical labeling method modification

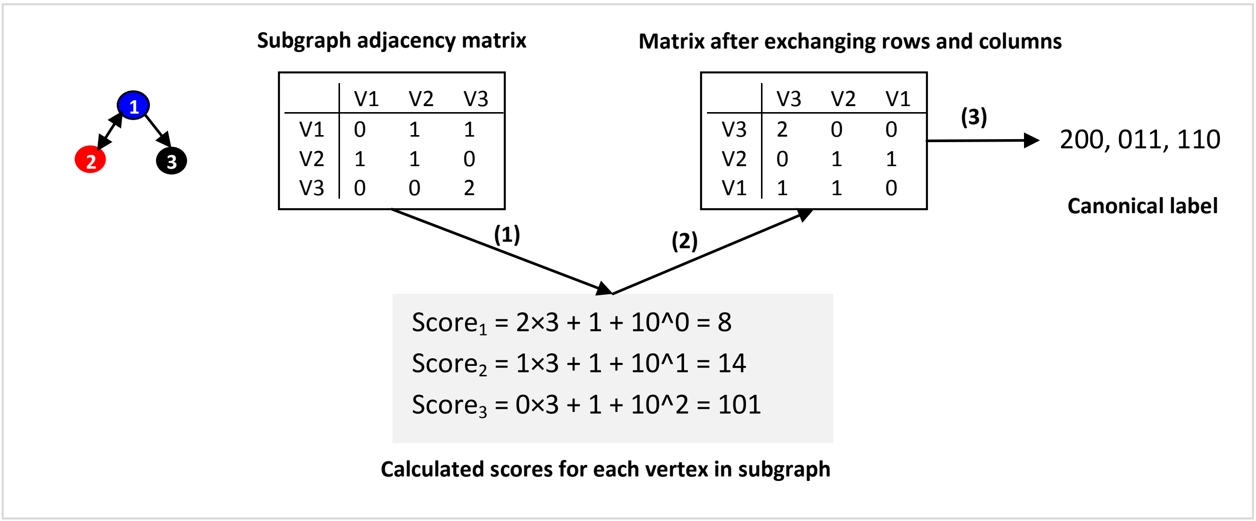
The published implementation of Subenum uses the following algorithm to compute canonical labeling of subgraphs:

```
for each vertex $v_i$ in subgraph *subg* do:
        Calculate $v_i$ 's score:
        $score_k$ = ( out_degree($v_i$) × K ) + in_degree($v_i$)     #K is subgraph size
end

sort scores in descending order

exchange adj-matrix rows based on sorted scores

exchange adj-matrix columns based on sorted scores
```

We modified Subenum's canonical labeling method to handle subgraphs with multiple node types. In order to do this, we changed the scoring formula to the following:

```
for each vertex $v_i$ in subgraph $subg$ do:
        Calculate $v_i$ 's score:
        $score_k$=(out_degree($v_i$)×K)+in_degree($v_i$)+ (10^($d_{ii}$))
end
```

This new scoring formula considers the node type in the calculation of canonical labeling. Supplementary Figure 4 shows how the score is calculated for an example subgraph.


Subgraph adjacency matrix
Matrix after exchanging rows and columns
1
2
3
(1)
(2)
(3)
200, 011, 110
Canonical label
Calculated scores for each vertex in subgraph

	V1	V2	V3
V1	0	1	1
V2	1	1	0
V3	0	0	2

	V3	V2	V1
V3	2	0	0
V2	0	1	1
V1	1	1	0

$Score_1 = 2×3 + 1 + 10^0 = 8$

$Score_2 = 1×3 + 1 + 10^1 = 14$

$Score_3 = 0×3 + 1 + 10^2 = 101$

Supplementary Figure 4: Example of modified canonical labeling computations
The sorted scores (V3, V2, and then V1) determines the order of row exchange and column exchange with the subgraph's adjacency matrix.

E. Search tree expansion algorithm modification

The published implementation of Subenum has high memory usage during the expansion of the search tree (Supplementary Figure 2), and as a result, runs out of memory when running on a personal computer with 4GB of RAM while processing networks with hundreds of thousands of edges. This is because Subenum expands nodes of the search tree using the Breadth First Search (BFS) algorithm. The BFS explores the immediate neighbor nodes first, before moving to the next level neighbors. In order to explore all subgraphs of size K using BFS, all of the vertices accessible through the root vertex in the previous K-1 levels of the tree are stored in memory. We addressed the high memory usage issue by replacing the BFS implementation with a Depth First Search (DFS) implementation. The DFS visits all of the descendant nodes of the current node first, and then backtracks to the next node in the graph. DFS provides a large memory savings without increasing computing time in the case of this algorithm, because the search is limited by definition to K-1 levels.

3) References

Megraw,M. et al. (2013) Sustained-input switches for transcription factors and microRNAs are central building blocks of eukaryotic gene circuits. Genome Biology, 14, R85.

Roy,S. et al. (2010) Identification of Functional Elements and Regulatory Circuits by Drosophila modENCODE. Science, 330, 1787–1797.

Shahrivari,S. and Jalili,S. (2015) Fast Parallel All-Subgraph Enumeration Using Multicore Machines. Scientific Programming, 2015, e901321.

Wernicke S. Vol. 3692. Proceedings of the 5th Workshop on Algorithms in Bioinformatics (WABI '05), Lecture Notes in Bioinformatics. 2005. A faster algorithm for detecting network motifs; p. 165-177.