From Integer Linear Programming to SAT-Solving in Computational Biology

Dan Gusfield

Plenary Talk Presented at The 2020 ALGO/WABI Conference
September 9, 2020
This work was supported by NSF grant 1528234.

Last Year at BCB/WABI

Almost exactly one year ago (Sept. 7, 2019) I gave a tutorial at the BCB/WABI conference on Integer Linear Programming in computational and systems biology.

To Recap:

Integer (Linear) Programming, abbreviated "ILP", is a versatile modeling and optimization technique, increasingly used in *computational and systems biology* in *non-traditional* ways.

ILP is often (but not always) very effective in solving *instances* of hard biological problems.

ILP Outreach

My book:

Integer Linear Programming in Computational and Systems Biology,

published last year, aims to convince people, particularly biologists, of the efficacy of ILP, and to teach them how to start using it to address computational problems in biology.

The book explores over fifty problems in computational and systems biology where ILP has been used, or surely could be used to solve practical *instances* of hard problems in computational biology.

Most of the problem instances I examined were successfully handled by ILP, but there were a few notable *failures*.

Also, In the Proposal for my Final NSF Grant

The last thing I said I would do was "Examine the efficiency of SAT-solving, compared to ILP, for problems in computational biology", since SAT-solving is not widely used in computational biology.

Of course, this was the kind of "I-hope-to" item that one rarely gets to, but with the help of two very talented undergraduate students (Hannah Brown and Lei Zuo) we actually did examine this question.

I fully expected to see that SAT-solving has no magic (and probably less) than ILP-solving has, to address hard computational problems. After all, *SATISFIABILITY* has very little structure compared to the beautiful and deep structure of *LINEAR ALGEBRA*.

But, contrary to my biased expectation, I learned something (never too old?), and I want to pass that on here.

Heresy, Hypocricy?

This talk is a somewhat *Heretical* at an Algorithms conference. ILP and SAT-solving are **Big, Crude Hammers** compared to the clever, nimble, artistic, even poetic algorithms presented at meetings such as ALGO and WABI. But it is easier to learn to hammer than to learn fine calligraphy.

Also, this talk highlights *deficiencies* of ILP, and yet I am (shamelessly) promoting my book on ILP - No hypocrisy - ILP in very valuable and effective in a huge number of problems in computational biology, but now we have a complementary tool: SAT-solving.

Three Types of Problems

We will look at three types of problems where SAT-solving looks increasingly better than ILP.

But first, I need to explain some basic things in case some of the participants are new to SAT.

CNF and SATISFIABILITY

A *CNF formula* contains *Boolean variables* (which can be set to either TRUE or FALSE); and a set of clauses, where each clause contains *negated* or *un-negated* variables, connected by OR relations. Example:

$$(X \vee \overline{Y} \vee Z) \wedge (\overline{X} \vee \overline{Z}) \wedge (Y \vee \overline{Z})$$

The formula is *satisfiable* if and only if there are (TRUE/FALSE) values for the variables that make the formula evaluate to *TRUE*.

ILP and SAT introduced through The Maximum Clique Problem

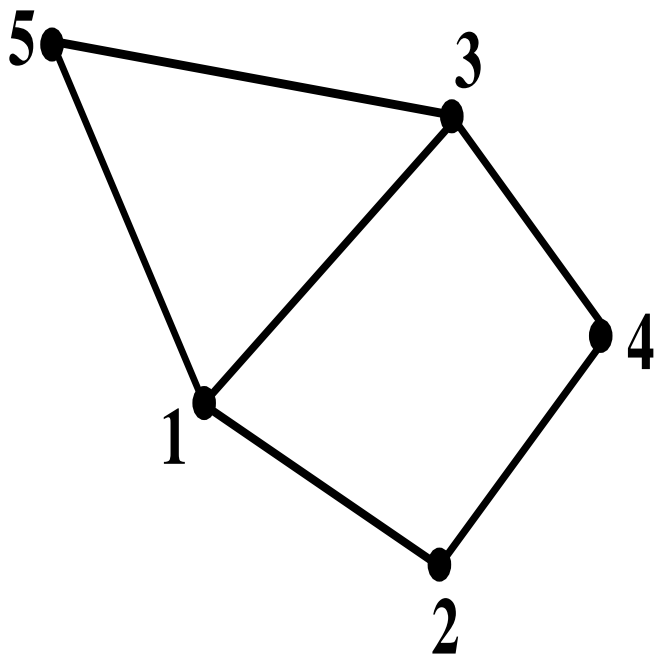

Figure : A clique K is a subset of nodes with an edge between every pair of nodes in K. This graph has a max-clique of size three.

The Max Clique Problem

Given an undirected graph G, *select* a set of nodes that forms a maximum-size clique in G.

The maximum-clique problem is a general combinatorial-optimization problem, but it actually has very broad and explicit application in computational biology. Many of the problems I discuss in my book involve cliques and near-cliques.

A Random Sample from PubMed Search using "Clique" in the Title

A too-long list - skip once you are convnced that cliques are important in computational biology.

- Functional **Cliques** in the Amygdala and Related Brain Networks Driven by Fear Assessment Acquired During Movie Viewing",
- **Cliques** for the identification of gene signatures for colorectal cancer across populations
- The Identification and Analysis of mRNA–lncRNA–miRNA **Cliques** From the Integrative Network of Ovarian Cancer
- Predicting interactions in protein networks by completing defective **Cliques**
- Combining Parallel Gibbs Sampling with Maximal **Cliques** for Hunting DNA Motif
- Identifying **Cliques** of Convergent Characters: Concerted Evolution in the Cormorants and Shags

More Cliques

- Exploring biological interaction networks with tailored weighted quasi-bi**Cliques**
- Predicting disease-related proteins based on **Clique** backbone in protein-protein interaction network
- Amino-Acid Network **Clique** Analysis of Protein Mutation Non-Additive Effects: A Case Study of Lysozme
- **Clique**-Based Neural Associative Memories with Local Coding and Precoding.
- Tumor stratification by a novel graph-regularized bi-**Clique** finding algorithm.
- Gene differential coexpression analysis based on biweight correlation and maximum **Clique**.
- Organizing principles of real-time memory encoding: neural **Clique** assemblies and universal neural codes.

Even More Cliques

- RCPred: RNA complex prediction as a constrained maximum weight **Clique** problem.
- Nested-**Clique** Network Model of Neural Associative Memory.
- Brain EEG Time-Series Clustering Using Maximum-Weight **Clique**.
- Protein-protein interactions prediction based on iterative **Clique** extension with gene ontology filtering.
- Estimating landscape carrying capacity through maximum **Clique** analysis.
- Identifying protein complexes in protein-protein interaction networks by using **Clique** seeds and graph entropy.

You Want More Cliques? Because I've Got Them

- Identifying protein complexes from interaction networks based on **Clique** percolation and distance restriction.
- Viral quasispecies assembly via maximal **Clique** enumeration.
- **Clique** topology reveals intrinsic geometric structure in neural correlations.
- **Clique**-Based Clustering of Correlated SNPs in a Gene Can Improve Performance of Gene-Based Multi-Bin Linear Combination Test.
- A novel protein complex identification algorithm based on Connected Affinity **Clique** Extension (CACE).
- Common pharmacophore identification using frequent **Clique** detection algorithm.
- Protein side-chain packing problem: a maximum edge-weight **Clique** algorithmic approach.

I've Got All the Best Cliques!

- Biased **Clique** shuffling reveals stabilizing mutations in cellulase Cel7A.
- Chemical structure elucidation from NMR chemical shifts: efficient data processing using bipartite matching and maximal **Clique** algorithms.
- Efficient similarity search in protein structure databases by k-**Clique** hashing.
- Protein threading with profiles and distance constraints using **Clique** based algorithms.
- Disease outbreak detection through **Clique** covering on a weighted ICPC-coded graph.
- Visualizing plant metabolomic correlation networks using **Clique**-metabolite matrices.
- **Cliques** I Have Known and Loved

Too Many Cliques

- **Clique**-based data mining for related genes in a biomedical database.
- Finding a Maximum Common Subgraph from Molecular Structural Formulas through the Maximum **Clique** Approach Combined with the Ising Model.
- **Clique**-finding for heterogeneity and multidimensionality in biomarker epidemiology research: the CHAMBER algorithm.
- Finding Friends in the Crowd: Three-Dimensional **Cliques** of Topological Genomic Domains.
- Interaction graph mining for protein complexes using local **clique** merging
- What to Expect When you Are Expecting **Cliques**

Back to ILP and SAT for Clique Finding

So, we want to use ILP and SAT-solving to *select* a set of nodes that form a clique in a graph.

How do we know that a set of selected nodes forms a clique?

A **Necessary** condition: For every pair of nodes i and j, if (i, j) is **not** an edge in G, then we **must not** select **both** nodes i and j.

The condition is also *sufficient*.

Fact: Any (non-empty) set of nodes that satisfies the above condition *will be* a clique in G. But, it need not be a maximum-sized clique.

Specifying a Clique with ILP

With the above NASC, we have:

The variables

One *Binary* variable, $C(i)$, for each node i of G.

Variable $C(i)$ indicates whether or not node i will be selected to be in a set K. If $C(i)$ is set to 1 we put node i into K; and if $C(i)$ is set to 0, we do not.

The ILP inequalities

For each *pair* of nodes (i, j) in G, we create the following inequality if (and only if) there is **no** edge in G between nodes i and j.

$$C(i) + C(j) \leq 1 \tag{1}$$

Specifying a Clique with CNF

With the above NASC, we have:

The variables

One *Boolean* variable, $C(i)$, for each node i of G.

Variable $C(i)$ indicates whether or not node i will be selected to be in a set K. If $C(i)$ is set to *True*, we put node i into K; and if $C(i)$ is set to *False*, we do not.

The CNF clauses

For each *pair* of nodes (i, j) in G, we create the following clause if (and only if) there is *no* edge in G between nodes i and j.

$$\overline{C(i)} \vee \overline{C(j)} \tag{2}$$

Good, But we Want a Maximum-Sized Clique

So, integer linear inequalities, and related Boolean clauses can easily *specify* the conditions for a set of nodes, K, to be a clique. But how do we find a *maximum-sized* clique?

For ILP, the size of the chosen clique is simply:

$$\sum_{i=1}^{i=n} C(i).$$

So to maximize the size of the clique, we simply add the *Objective Function:*

$$\text{Maximize} \sum_{i=1}^{n} C(i),$$

and let an ILP-solver rip.

Using SAT-solving to Count the Nodes in a Clique

If we generate the CNF clauses described above, and pass them to a SAT-solver, it will set some $C(i)$ variables to True, so that their associated nodes must form a clique. But how big will that clique be? That is (equating "True" with "1" and "False" with "0") what will $\sum C(i)$ be?

Using CNF formulas and SAT-solving, it is **not** as simple to determine the size of the chosen clique as it is in ILP.

But, we can't maximize $\sum C(i)$ if we can't determine it. This "counting problem" is the **Big Achilles Heel** of SAT-solving.

Still, there are several ways to use SAT to count. Here is the (simple) approach I generally use:

Using CNF to Count the Nodes in a Clique

Let n denote the number of nodes in G. *Think* of counting the number of $C()$ variables set True by doing a sequential *scan* through the variables, examining the value of each $C(i)$, for i from 1 to n. But, we can't actually specify such a *sequential* scan in CNF. Instead,

For each i from 1 to $n+1$, create i Boolean variables, $T(i,d)$, where d runs from 0 to $i-1$. So for $i=1$, there is only one variable, $T(1,0)$; and for $i=n$, there are n variables $T(n,0),...,T(n,n-1)$.

We interpret $T(i,d)$ as the proposition:

> **At least d** *variables in the set* $\{C(j) : j < i\}$ *are set* **True**.

Note the phrase **"At least"**. Do not confuse it for **"Exactly"**.

The Clauses that Count

Remember that $T(i, d)$ represents the proposition:
At least d variables in $\{C(j) : j < i\}$ are set True. So to begin, for each i, we create the single-variable clause:

$$T(i, 0),$$

which forces variable $T(i, 0)$ to be True, and means that *at least zero* of the $C(j)$ variables, for $j < i$, are set True (duh!).

More Clauses that Count

We also create clauses that say:

$$T(i,d) => T(i+1,d),$$

which in CNF is

$$T(i+1,d) \vee \overline{T(i,d)},$$

meaning that *if* at least d variables, $C(j)$, for $j < i$ are set True, *then* at least d variables, $C(j)$, for $j < \mathbf{i{+}1}$, are set True (again, duh!).

(Recall that $A ==> B$ is equivalent to $B \vee \overline{A}$)

Clauses the Really Count

Now the important clauses: For each of the variables, $T(i, d)$, for i from 1 to n, we need a clause to implement:

> *IF* $T(i, d)$ *is True, AND* $C(i)$ *is True, THEN* $T(i + 1, d + 1)$ *must be True.*

which is equivalent to the CNF clause:

$$T(i + 1, d + 1) \vee \overline{T(i, d)} \vee \overline{C(i)}.$$

Are we There Yet? No!

With these clauses: *If* $\sum C(i) \geq d$, *then* $T(n+1, d)$ *will* be set True.

But, what keeps $T(n+1, d)$ from being set True even when $\sum C(i) < d$?

For that direction, we need:

For each i from 1 to n:

$$(*) \quad T(i+1, d) => T(i, d) \; OR \; [T(i, d-1) \; AND \; C(i)].$$

Say it in CNF

Boolean formula (*) is equivalent to:

$$(**) \quad T(i,d) \ \vee \ [T(i,d-1) \ \wedge \ C(i)] \ \vee \ \overline{T(i+1,d)}$$

But, the second phrase has a $\wedge$ in it, so is not in CNF. What to do?

How do we Transform $[T(i, d-1) \wedge C(i)]$ to CNF?

By *Tseytin encoding*, we can create a new Boolean variable $Q(i, d-1)$, and create three short clauses that make $Q(i, d-1)$ equivalent to $[T(i, d-1) \ \wedge \ C(i)]$.

$$\begin{aligned} T(i,d) \vee \overline{Q(i,d-1)} \\ C(i) \vee \overline{Q(i,d-1)} \\ \overline{T(i,d)} \vee \overline{C(i)} \vee Q(i,d-1) \end{aligned} \tag{3}$$

Then, we can implement (**) in CNF as

$$T(i,d) \ \vee \ Q(i,d-1) \ \vee \ T(i+1,d).$$

Maximizing, using a SAT-solver

We have established that $\sum C(i)$ will be equal to the *largest* d such that $T(n+1, d)$ is set True.

So, if we have a target t and want to **test if** there is a clique in G of size at least t, we simply add the single-variable clause:

$$T(n+1, t),$$

and let a SAT-solver determine if the CNF-formula is *satisfiable* or not.

With the ability to *test* if there is a clique of size (at least) t, we can find a *maximum-size* clique by forming and testing a series of CNF formulas where t is varied. Binary search is possible, but not advised! Why?

Live Demo - I hope!

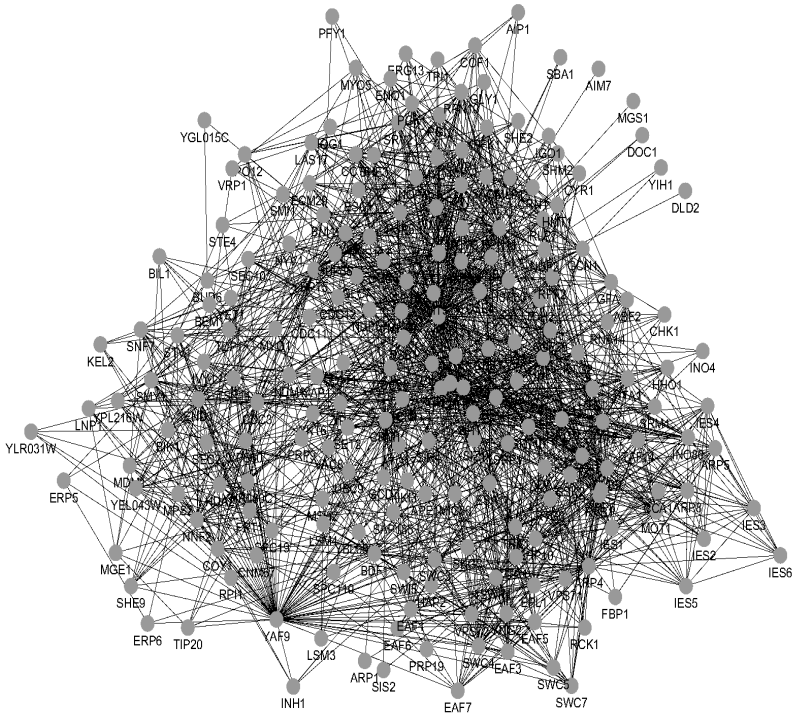

Figure : Yeast Protein-Protein Interaction Graph (hairball) with only 209 nodes and 1776 edges. Can you spot a Max-clique by eye? Both ILP and SAT-solving found a clique of size 12 and proved that it is maximum, in about one second. That is very impressive to me.

Results in a Random Graph with One-Thousand Nodes

In a typical example, a random graph with one-thousand nodes and edge density of 0.2 was generated.

The ILP for maximum clique in this graph found a clique of size 7 after about 4.5 minutes, at which point the upper-bound was about 42.

Then after about *three* more *hours*, the largest clique found was still 7, and the upper bound 33.

After *five hours and five minutes*, Gurobi found a clique of size 8, and the upper bound was fifteen. After *five hours and 30 minutes*, Gurobi terminated with an optimal, maximum-sized clique of size 8.

SAT-Solver Results on This Graph

Tests with a target of 6 by three different SAT-solvers (CryptominiSAT5, Glucose-Syrup, pLingeling) found a clique of size 6, in 4 seconds, 11.7 and 8 seconds, respectively.

Tests with a target of 7 found a clique of size 7, in 9 seconds, 12 seconds, and 11.5 seconds, respectively.

Tests with a target of 8 found a clique of size 8, in 38 minutes, 55 seconds (and again in 10 minutes), and 43 minutes, respectively.

Tests with a target of 9 found the problem UNSATISFIABLE in 2736, 821 and 9807 seconds, respectively.

An Integrated Run

Starting with a target of three and incriminating by one until an UNSATISFIABLE target was found, the SAT-solver (Glucose-Syrup) determined that the maximum-size clique is of size 8, in about 28 minutes. Most of the time was for the UNSAT target of 9.

Compare the SAT-solver time to the five hour 30 mins. time for ILP. Hence, in this larger test, SAT-solving won.

Problem 2: Transforming Gene Order by Reversals

There are important biological phenomena that occur at a scale larger than individual nucleotides, e.g. at *chromosomal* or *genomic* scales. For example: *long chromosomal reversals (inversions)*.

If we represent each gene by a distinct integer, the interval with ten genes: 1 10 4 5 2 6 3 9 8 7 becomes: 1 10 6 2 5 4 3 9 8 7 when the interval containing 4 5 2 6 is reversed.

The computational problem of interest is:

The Sorting-by-Reversals Problem: Given two permutations, P_1 and P_2 of the integers 1 to n, find the *minimum* number of interval reversals that transforms P_1 to P_2.

Turnips to Cabbage Example

In the classic **Turnips to Cabbage** dataset with ten genes, the problem is to transform

1 10 4 5 2 6 3 9 8 7

to

1 2 3 4 5 6 7 8 9 10

The optimal sequence requires four reversals:

1 10 4 5 |2 6| 3 9 8 7

1 |10 4 5 6 2 3 9 8 7|

1 |7 8 9 3 2| 6 5 4 10

1 2 3 |9 8 7 6 5 4| 10

1 2 3 4 5 6 7 8 9 10

The Signed Variant

In the *signed* variant of the problem, each integer in P_1 has a positive or negative sign, and all integers in P_2 have a positive sign.

Whenever an interval is reversed, the sign of each integer in the interval changes to the **opposite** sign. Then, the goal is to transform P_1 to the all-positive P_2, using the minimum number of reversals.

Algorithmic Status

The (unsigned) Sorting-by-Reversals problem is NP-hard.

But (amazingly), there are *polynomial-time* algorithms for the **signed** version of the problem! These algorithms are complex, and are the result of deep and hard thinking over many years by top algorithmists. I consider them subtle and poetic. We should not expect the average algorithmist or biologist to come up with such algorithms.

So, what can the *Big Hammers*, which are tools for the masses, do? It is easy to formulate the unsigned and signed versions of the problem, using either ILP or SAT. This involves a fairly straight-forward translation of the meaning of a tranformation, into integer inequalities (see chapter 18 of the book) or into CNF clauses. Lei Zuo implemented and explored the SAT-solving approach.

First Results

Gurobi 8.1 solved the *Turnips to Cabbage* problem in fifteen seconds, and the SAT formulation was generated and solved in five seconds.

In a second classic dataset, the **Field Mustard to Black Mustard** dataset of length twelve, the ILP formulation was solved by Gurobi 8.1 in about 102 minutes, but earlier in only 33 minutes by Gurobi 8.0 (go figure!).

The SAT approach solved the Field Mustard to Black Mustard problem in about 3 minutes, 25 seconds. So again, a win for SAT-solving.

Random Data

Results from the examination of random permutations of length up to eight is consistent with the tests on plant genomes.

Table : Times (rounded to the nearest tenth of a second) taken by Gurobi and pLingeling with ten randomly generated unsigned sequences of length eight

Sequence number	1	2	3	4	5	6	7	8	9	10
pLingeling	1.3	3.1	2.5	1.3	1.5	1.7	2.8	1.6	12.9	1.3
Gurobi	5.0	13.5	11.2	9.0	6.2	10.5	9.1	3.0	74.0	8.7

The Signed Variant

We also compared running times for the ILP and SAT approaches to the **signed** variant. There, the overall running times were longer for both problem variants, but more interestingly, there were *substantial* differences in the times:

Table : Times (rounded to the nearest second) taken by Gurobi and pLingeling with ten randomly generated signed sequences of length eight

Sequence number	1	2	3	4	5	6	7	8	
pLingeling	57	463	56	11	50	430	54	58	1
Gurobi	30381	36387	7095	7855	3294	62050	6282	22293	3

Where are We?

So, we have seen one problem *Maximum Clique* where SAT-solving performed better than ILP, for *large* graphs, but ILP was still of use.

We then saw *The Unsigned Gene Reversal Problem* where SAT-solving was faster, but ILP was still impressive. But in the *Signed Gene Reversal Problem* ILP was *significantly* slower than SAT-solving.

Next, we will see two related problems where ILP was essentially useless, but SAT-solving performed acceptably.

Problem 3: Evolutionary Trees and Phylogenetic Networks

Trees are the traditional way to think about and represent evolution, due to branching, but trees are often too simplistic, both for biological and methodological reasons.

Instead, there a now many evolutionary problems that are framed in terms of finding **networks** (usually DAGs) that represent more complex evolution than trees do.

But, often the networks are required to contain known trees, or reflect partial information about trees.

Directed Acyclic Graphs (DAGs) and Trees

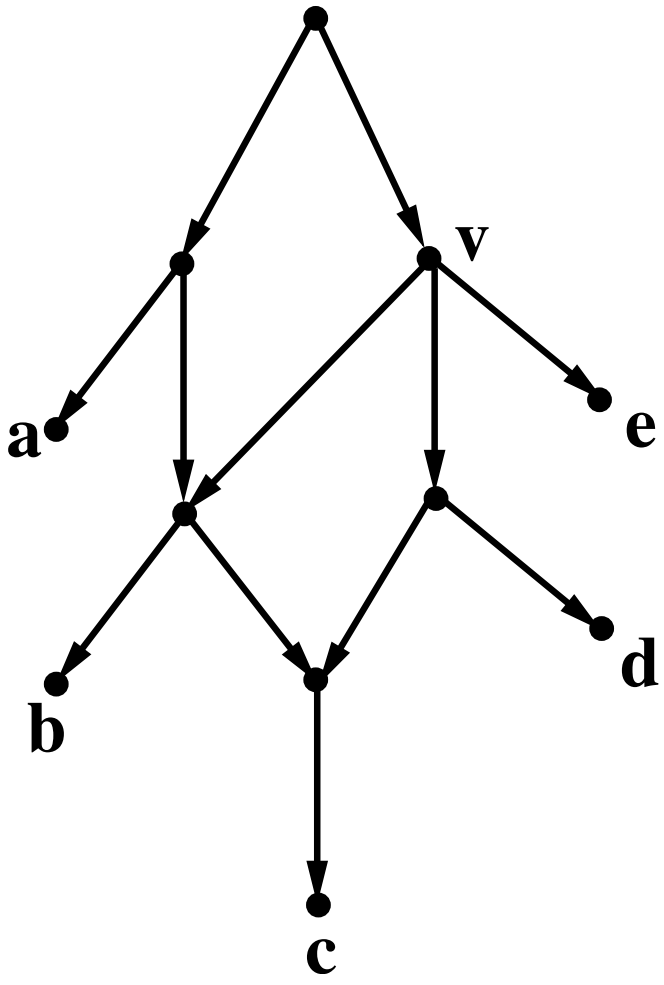

Figure : DAG D contains four subtrees that reach all of the leaves.

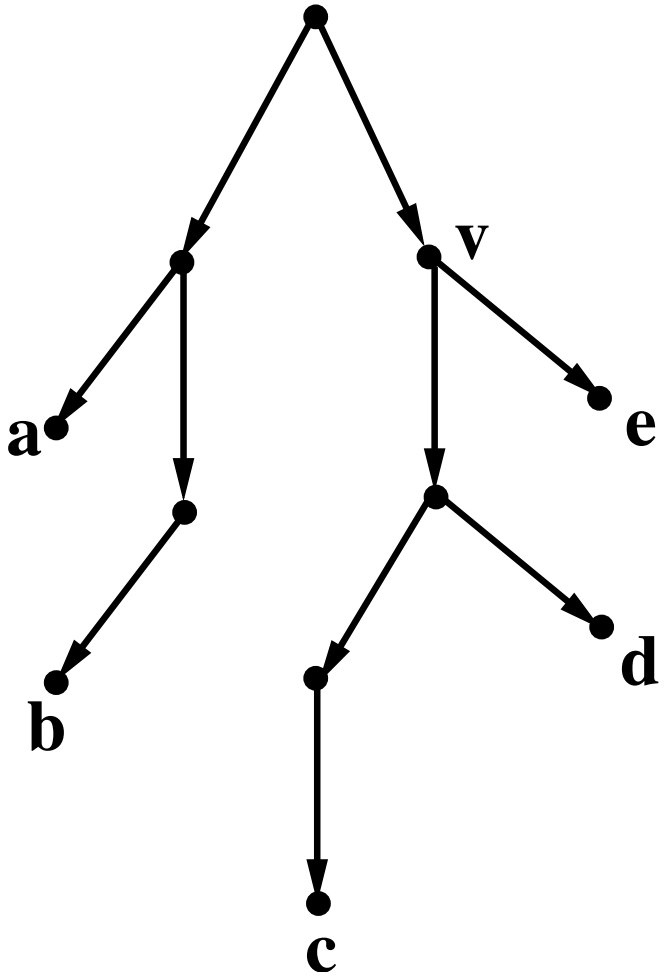

Figure : A subtree of D that reaches all leaves. The leaves reached from node v are $\{c,d,e\}$.

The Central Definition

Let S be a set of n *leaf* labels.

A *Directed Acyclic Graph (DAG)* D with n leaves labeled S, **displays** a subset s of S if:

a) There is some tree T_s embedded in D that reaches *all* of the leaves of D, and

b) There is a node v in T_s, where the leaves of T_s that are reachable from v have *exactly* the labels in s.

Such a subset s is called a *cluster*.

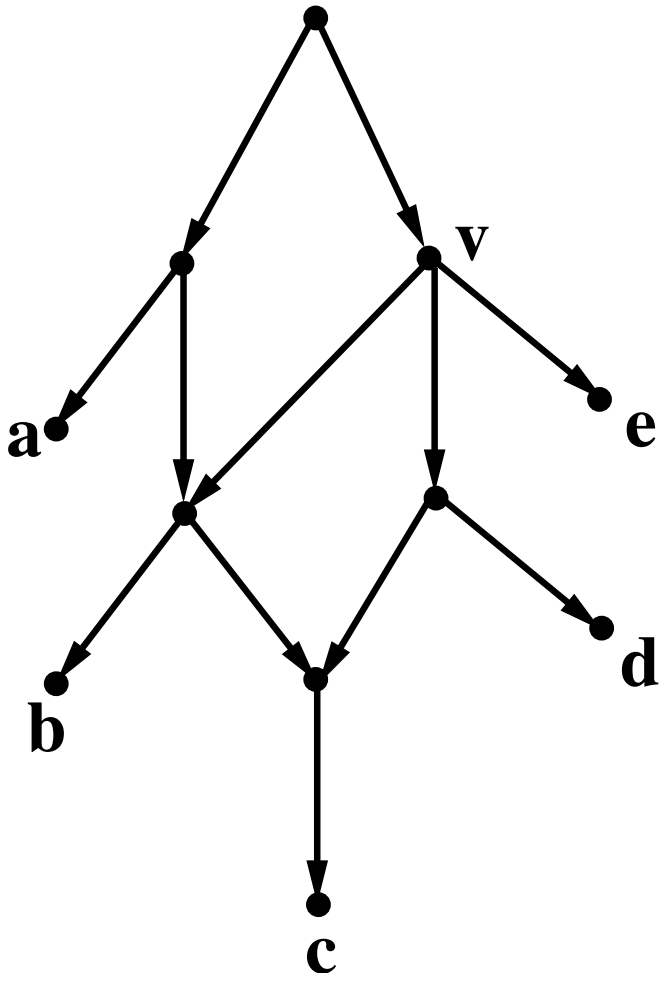

Figure : DAG D.

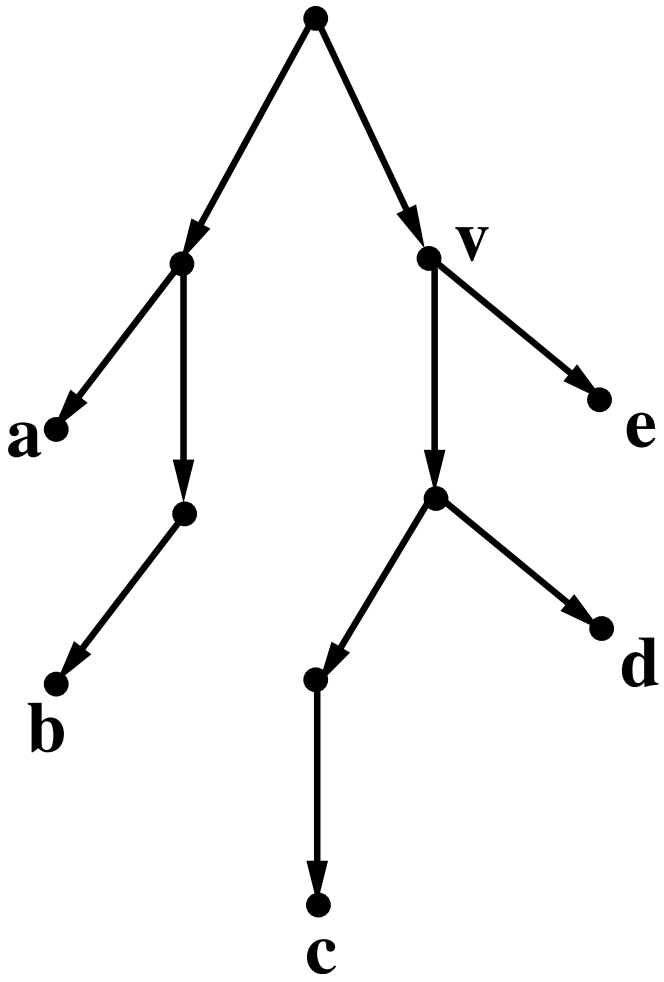

Figure : The tree displays cluster $\{c, d, e\}$, and other clusters.

Reticulation Networks

As above, let S be a set of labels, and now let F be a **family** of *subsets* (clusters) of S.

A DAG that displays **every** cluster of F is called a **Reticulation Network** for S, F.

Any node in a reticulation network with in-degree more than one is called a *reticulation node*.

Reticulation Networks

For example, let $S = \{a, b, c, d, e\}$, and $F = \{a, b\}, \{b, c\}, \{d, e\}, \{b, c, d, e\}$.

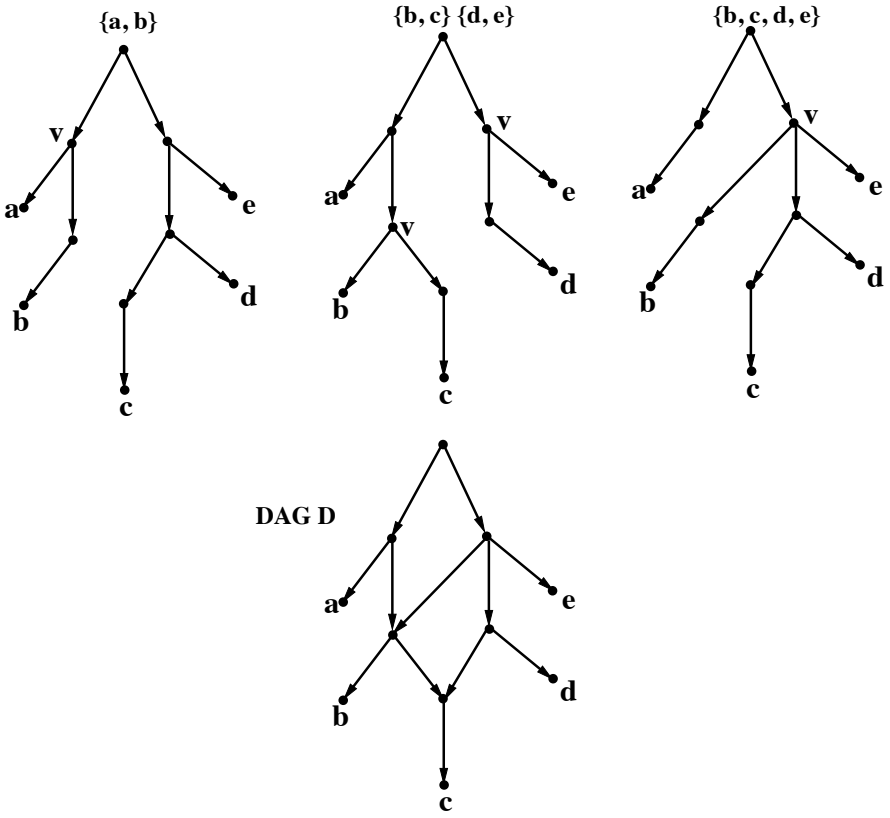

Figure : DAG D, displaying the shown subsets of {a,b,c,d,e}.

Minimum-Reticulation

The *Minimum-Reticulation Problem*: Given S and F, construct a reticulation network for S, F, with the *fewest* reticulation nodes, over all reticulation networks for S, F where every reticulation node has in-degree two.

Solving Minimum-Reticulation with ILP My PhD student (now, Dr. Julia Mastieva, PhD) and I, developed three different ILP formulations for this problem, but *none* of them could solve anything more than trivial problem instances.

The ILP solver would typically find OK, but not optimal DAGs that displayed the clusters in F. And even when they had an optimal solution in hand, there would be a huge gap between the solution value and the lower bound that the solver found.

I *conjecture* that this is a somewhat general problem that ILP has for *network design* problems.

Solving Minimum-Reticulation with SAT

So, Hannah Brown and I explored SAT-solving for this problem. The CNF formulation we developed is a fairly straight-forward translation of the problem definitions into CNF. The resulting CNF formulation is involved (45 types of clauses), but not surprising.

Again, the *counting problem* was the most difficult aspect of the implementation.

One thing that helps: there is a fast *Upper Bound* on the minimum number of reticulations needed, computed by program SHRUB (developed at UC Davis by Yun Song). We use that to choose the first target number of reticulations to test.

Prior, Related Work

Before discussing empirical results for the Minimum-Reticulation problem, there is a related problem that was tackled earlier by SAT-solving in a paper (Ulyantsev and Melnik, ALCOB 2015).

In that problem, the input is not a set of *clusters*, but a set of *trees*:

Hybridization Network Problem: Given a set of rooted *binary* trees, $\mathcal{T}_1, \mathcal{T}_2, ..., \mathcal{T}_k$, each with the same set of n labeled leaves, construct a DAG D with a single root and with each node of in-degree at most two, that displays each of the trees $\mathcal{T}_1, ..., \mathcal{T}_k$, *minimizing* the number of reticulation nodes.

Results

They showed that they can successfully solve most of the problem instances they explored, with up to five binary trees and n up to 40, and with the number of reticulation nodes up to 13. However, their method was not *pure SAT-solving*. It also used combinatorial insights developed for the Hybridization problem.

Their CNF-formulation is very complex, with 68 different *types* of clauses, and is limited to binary trees.

Can we Reduce the Hybridization Problem to the Reticulation Problem?

No, and Yes.

The first attempt is to take each input tree $\mathcal{T}_i$, and create the set of $n-1$ *clusters*, each defined by a distinct non-leaf node v in $\mathcal{T}_i$. Then this collection of clusters can be used as input to the Minimum Reticulation Problem.

The resulting DAG will display all of the clusters in the input trees, *but* it will **not** be guaranteed to display the actual trees $\mathcal{T}_1, ..., \mathcal{T}_k$.

An Easy Fix

Recall that in the Minimum Reticulation Problem, for each input cluster $s \subset S$, a SAT solution must specify a directed tree T_s containing all n of the S-labeled leaves; and T_s must have a node v that reaches *exactly* the leaf labels in s.

To use Minimum-Reticulation to solve the Hybridization-Network Problem, we change the CNF to enforce the following:

> *If two clusters s and s' come from the same input tree $\mathcal{T}$ (in the Hybridization problem), then the created trees, T_s and $T_{s'}$, specified by a SAT solution (to the Reticulation problem), must be identical.*

These forced equalities are actually more easily implemented than what is suggested by the above statement. (Homework problem)

Why Does this Work?

The Fundamental Theorem of Trees:

> *The set of clusters derived from a directed tree T uniquely specifies T.*

So, when *all* of the clusters from an input tree $\mathcal{T}$ must use the *same* tree T in the created DAG D, *T must be $\mathcal{T}$*.

Take Home Quiz: Why doesn't the first attempt work?

Some Empirical Examples

data	#trees, #leaves, UB	target	RET	HYB
3PhytRpocits	3, 20, 9	9	1.33s	57s
		8	42s	5m 8s
		7	33s	7m 26s
		6	T 1000s	T 2000s
2NdhfWaxy	2, 20, 7			
		9		58s
		8		42s
		7	1.8s	55s
		6	30s	52s
		5	T 3000s	8m 18s UNSA
3NdhfPhytWaxy	3, 15, 4			
		6		0.4s
		5		2.99s
		4	0.8s	17.8s
		3	T 31,000s	9m 4s UNSAT

Table : Results of experiments with different datasets.

Results

ILP is essentially useless for these network problems. For the first dataset above, ILP found a feasible solution with 12 reticulations, and a lower-bound of **zero**, and was terminated after three hours.

This Reticulation approach to the Hybridization-Network Problem is simpler and more flexible than the SAT approach in the 2015 paper. It allows the input of *non-binary* trees; or *hybrid* problems more constrained than the Min Reticulation problem, but less constrained than the Hybridization-Network Problem; or problems where the clusters don't come from trees. But, the SAT formulation still has forty-five types of clauses.

Running time comparisons of the alternative CNF-formulations are not possible using the earlier software, because it uses non-SAT algorithmic insights to simplify each problem instance.

Lessons Learned

- SAT-solving for optimization problems works well for problems with *binary* variables, and where the value of an optimal solution is an integer; and where the range of possible solution values is small.

  Those conditions are common in optimization problems that arise in computational biology. So SAT-solving is attractive for computational biology applications. But, ILP is attractive for a much wider range of conditions, particularly when the range of solution values is large.
- UNSAT is much harder than SAT.
- Debugging CNF is really, miserably hard. I blame DIMACS. ILP is much easier to debug.
- Similar problems and similar formulations can have very different empirical behaviors - both in ILP and in SAT.

Lessons Learned

- Expect little or incorrect documentation for SAT-solvers. It took me months of search to even find a list of the 300+ options in Lingeling. I finally found a list, but no explanations. It's the wild-west compared to ILP.
- Running times for SAT-solvers have *Huge* variance. ILP solvers have very low variance in comparison.
- There is no "best" SAT-solver - huge variance again, based on the type of problem. Contrast this to the uniform dominance of Gurobi and Cplex compared to other ILP-solvers.
- Several "obviously slam-dunk" ideas for speeding up SAT-solving failed miserably. Why?
- The "beauty vs strength" debate for what makes ILP solve fast is unresolved - much conflicting evidence, and some misleading theory. I don't accept the current "wisdom" that strength is better than beauty.

Final Cleanup

We have seen problems where SAT-solving is more effective than ILP-solving. And this illustrates the **main point** of this talk:

> *SAT-solving is another powerful computational tool that should be explored and exploited in computational biology.*

However, the particular problems we explored were *selected* because ILP efforts had not worked as well as desired. For most of the problems considered in my book, and many others in the literature, ILP is a very successful tool. And, in this project we also looked at a problem (protein folding under the HP model) where ILP did *better* than SAT; and at another problem where neither SAT nor ILP worked well (History Bound using Networks). (See our paper, Brown et al, at ALCOB 2020, in LNCS 12099).

Thank You

Questions?