Integer Linear Programming in Computational and Systems Biology

Dan Gusfield

Presented at The 10th ACM Conference on Bioinformatics, Computational Biology, and Health Informatics (ACM BCB), Niagara Falls, September 7, 2019

This tutorial is adapted from the the book *Integer Linear Programming in Computational and Systems Biology: An entry-level text and course*, published by Cambridge University Press, 2019

Why Integer Programming?

Integer (Linear) Programming, abbreviated "ILP", is a versatile modeling and optimization technique.

Increasingly used in *computational and systems biology* in *non-traditional* ways.

Often very effective in solving *instances* of hard biological problems.

Moreover, even for a problem where a worst-case efficient general algorithm might be possible, the time and effort needed to find it, and the time and effort needed to implement it as a computer program, are typically much greater than the time and effort needed to formulate and implement an ILP solution to the problem.

Why ILP?

Highly engineered, commercial ILP solvers are available (now free to academics and researchers) to solve ILP formulations.

The improvement of the best solvers has been *spectacular*, with an estimate that (combined with faster computers) benchmark ILP problems can now be solved 200-*billion* times faster than twenty-five years ago.

Exploiting ILP, some biological problems of importance can be modeled in a way that allows a solution in seconds on a laptop, while more common models require days, weeks or months of computation on large clusters.

Why learn from me?

I don't know much, but I seem to get the job done, so you really don't need to learn much, and I will keep it simple.

A Fly-Over Introduction to Integer Linear Programming

First: Linear Programming (LP) and its use. Three parts:

A) The *concrete formulation* of a Linear Program (or model), given all the data required to specify a specific problem instance.

B) The *solution* of a concrete formulation.

C) The *abstract formulation* of a Linear Program.

The Threatened-Species-Protection Problem

A real problem concerning conservation of endangered plant species in the *Cape of South Africa*, which is

> *... one of the most botanically species-rich areas of the world with more than 9000 species ... [**?**]*

These species span more than 700 *genera*, and at least 274 of them contain one or more species that is classified as "vulnerable, endangered, or critically endangered".

Conservation agencies have (limited) resources to preserve *some* areas of the Cape, and hence to help protect some of the threatened species. So, the general question is how to most effectively use the available resources to protect threatened species.

A simple variant of the problem

The Cape is divided into about 200 square regions. It is assumed that in each region, we know the abundance of each of the 274 threatened species. We measure abundance of a species in a region by the area occupied by that species.

The cost to preserve *all* the land in any region is also assumed to be known, but partial preservation of a region is also possible.

Also, for each threatened species, a *conservation target* is established, meaning that the species would be considered protected if the *total area* occupied by the species in the preserved land (over all the regions) contains more than its conservation target.
Then, the first general problem is

> **The Species-Protection Problem** *What is the least expensive way to protect all of the threatened species?*

A toy example

Region	Area Occupied by α in square Kilometers	Area Occupied by β in square Kilometers	Cost for Preservi
A	24	83	\$97
B	36	11	\$73
C	0	29	\$22
D	15	0	\$11
E	40	18	\$45
Target	64	87	

Table : Concrete data for an instance of the Species Protection Problem

The concrete problem instance

We *formulate* a *linear program (LP) model* (also called an *LP formulation*) that describes (or expresses) the details of the *specific* problem instance. That formulation, with all the details of the problem *instance*, is called a *concrete* LP formulation.

To solve the concrete instance, we *solve* the concrete LP formulation using an LP *solver*. The solution will specify how much of each region to preserve, and what the total cost will be.

ILP variables

To create an LP formulation (model) for a concrete problem instance, we begin by creating linear programming *variables*. An LP variable can take on a *numerical value*. The LP variables for

the *species protection problem* express the unknown values that we ultimately want to determine. X_A denotes the *fraction* of region A

that will be preserved. Being a fraction, X_A can only take on a value between 0 to 1 (inclusive).

The variables X_B, X_C, X_D and X_E have analogous meanings for regions B, C, D and E, respectively.

Constraints

The next step in formulating a concrete LP model for a problem instance is to develop *linear constraints*, which are either *inequalities* or *equalities*. The inequalities and equalities express the additional constraints on the values that are *permitted* to be assigned to the variables.

A *linear function* of a set of variables is formed by multiplying each variable by a specific *coefficient* (or constant number) and adding together the resulting terms.

$$3X_A + 4X_B = 17$$

is a linear equality.

$$3X_A + 4X_B \leq 13$$

is a linear inequality.

The toy example

In the concrete formulation for the toy instance of the Species Protection Problem, we have the simple, initial constraints:

$$\begin{aligned} X_A &\leq 1, \\ X_B &\leq 1, \\ X_C &\leq 1, \\ X_D &\leq 1, \\ X_E &\leq 1. \end{aligned} \tag{1}$$

More interesting constraints

The more interesting, and complex constraints come from the requirement to protect both of the threatened species. The following constraints express what is needed to protect species α and β:

$$\begin{aligned} 24X_A + 36X_B + 0X_C + 15X_D + 40X_E &\geq 64 \\ 83X_A + 16X_B + 19X_C + 0X_D + 18X_E &\geq 87. \end{aligned} \tag{2}$$

Feasible solutions

Some combinations of values for the variables X_A, X_B, X_C, X_D *satisfy* (make true) all the inequalities, but some combinations of values *violate* one or more of the inequalities. A combination of values assigned to the variables that satisfies *all* of the inequalities is called a *feasible solution*.

If there are no feasible solutions, then the set of constraints is called *infeasible*.

For example, if we set all five variables to 0.66, then the inequalities are satisfied, and the sums in the inequalities are 75.9 and 89.76, respectively.

However, if we set all the variables to 0.6, then the first sum becomes 69, and the second sum becomes 81.6. In that case, the second inequality is violated, hence, that assignment of values is *not* a feasible solution.

The Objective function

So far, we have not used the *costs* required to preserve different regions, yet the stated problem is to protect both of the threatened species, spending the *minimum* (i.e., least) amount of money possible. How is that objective included in the model?

When values are assigned to the five X variables, the total cost (in thousand of dollars) will be equal to:

$$97X_A + 73X_B + 22X_C + 11X_D + 25X_E. \tag{3}$$

So, the total cost is a *linear function* of the five X variables. Therefore, the *objective function*, which expresses the goal of spending the least money possible (while protecting both species) is stated as:

$$\text{Minimize} \quad 97X_A + 73X_B + 22X_C + 11X_D + 25X_E \tag{4}$$

The concrete linear programming formulation

Summarizing, the full concrete LP formulation for the toy problem instance is shown below.

$$\text{Minimize} \quad 97X_A + 73X_B + 22X_C + 11X_D + 25X_E$$

Subject to the constraints:

$$24X_A + 36X_B + 0X_C + 15X_D + 40X_E \geq 64$$

$$83X_A + 16X_B + 19X_C + 0X_D + 18X_E \geq 87$$

$$X_A \leq 1$$

$$X_B \leq 1$$

$$X_C \leq 1$$

$$X_D \leq 1$$

$$X_E \leq 1$$

Feasible and Optimal Solutions

A *feasible solution* to a concrete LP formulation is an assignment of values to the variables that satisfies all of the constraints. However, a feasible solution does *not* need to be one that

minimizes the objective function. A feasible solution the minimizes the objective function is called an *optimal solution*.

For example, when all the variables are given the value 0.66, the value of the solution is 150.48. As we will see later, this feasible solution is *not* an optimal solution, because there is a feasible solution with objective value 108.7.

LP solvers for concrete LP formulations

Once created, the concrete LP formulation can be input to an *LP solver* (in the proper format). If there is a feasible solution to the concrete LP formulation, the LP solver will determine and report an *optimal* solution.

Alternatively, if there is *no* feasible solution to the concrete LP formulation, the LP solver will determine and report that; and if there is a feasible solution, but there is no bound on the value of the feasible solutions (essentially infinity for maximization problems, or negative infinity for minimization problems), the LP solver will determine and report that fact.

The concrete optimal solution

In the concrete LP formulation for the toy Species Protection Problem has optimal objective value (after rounding up) of 108.7, which is achieved by setting (after rounding up) X_A to 0.831, X_D to 0.269, X_E to 1, and the other two variables to 0.

So, in this solution, none of regions B and C will be preserved, all of region E will be preserved, and regions A and D will be partly preserved.

What If?

One of the most useful features of linear programming is the ease in which *"what if"* questions can be explored, once the first LP formulation is created and solved.

Suppose there is pressure to specifically preserve some land in region B. From the optimal solution to the concrete formulation above, we know that both of the threatened species can be protected for $108.7K$ dollars. In that solution, *none* of region B was preserved.

It *might* still be possible to preserve *some* of region B, while protecting both species α and β, and spending no more than in the original optimal solution.

What If?

So, we can ask:

> *How much of region B can be preserved, without spending more than \$108.7K, while still protecting both species α and β?*

To answer that, we change the objective function to:

$$\text{Maximize } X_B,$$

and add the constraint:

$$97X_A + 73X_B + 22X_C + 11X_D + 25X_E \leq 108.7.$$

Then we use the LP solver to find an optimal solution to the modified concrete LP formulation. Running the solver, we get a new optimal solution with objective value of 0.00296.

That means it is *not* possible to preserve more than a very small amount (less than one third of one percent) of region B, without increasing the total amount spent for land preservation.

Another What-If

Given that very little of region B can be preserved (while protecting both α and β) without increased spending, we could next ask:

> *How much would we have to spend if we want to preserve at least 10% of region B, and of course, protect both species α and β?*

The answer is to start with the first LP formulation and add in the constraint:

$$X_B \geq 0.1.$$

Solving this concrete LP formulation results in an optimal solution with objective value of $111.7K$ dollars, an increase of only $3K$ dollars (a steal!).

LP-Solvers

LP-algorithms (e.g. simplex method, interior point method).

When the details of an LP-algorithm are written into an executable computer program, the program is called an *LP-Solver*.

An LP-Solver takes in a concrete LP formulation (in some, usually user-unfriendly rigid format), and returns the value of the optimal solution, together with values assigned to the LP variables in the solution.

Solvers I have used:
Gurobi Optimization
IBM Cplex
GLPK

Integer Linear Programming (ILP)

Linear Programming allows the LP variables to be given *fractional*, i.e. non-integer, values. *Integer* Linear Programming (ILP, MILP) simply refines Linear Programming by *requiring* that (some of) the variables in a formulation only be given *integer* values.

In the Species Protection Problem, this means that a region must either be completely preserved, or not at all. This can have a large effect on the optimal solution.

The optimal solution to an LP formulation (removing the integrality requirements) will be no worse, and is often strictly better, than the integer optimal solution.

An ILP formulation where the variables are further constrained to only take on values of *zero* or *one*, is called a *binary* formulation.

ILP-Solvers

All of the LP-solvers discussed earlier, Gurobi, Cplex, GLPK, are also ILP-solvers. SCIP is another ILP solver.

At the high level, the algorithms that ILP-solvers use to find an integer optimal solution are quite *different* from the algorithms used to solve LP formulations. But ILP methods usually require creating and solving *many* concrete LP formulations.

Part II. Biological Networks and High-Density Subgraphs

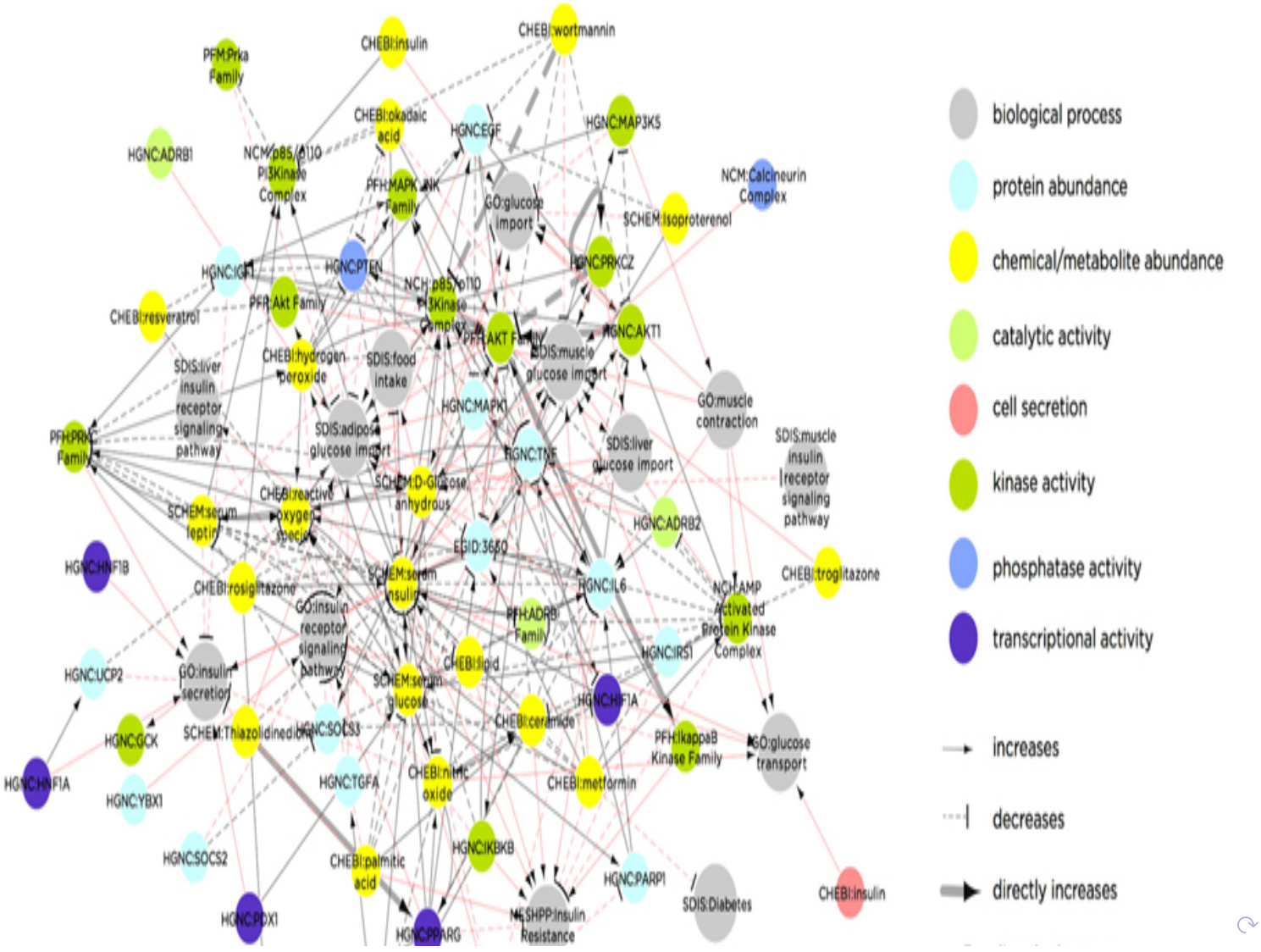
A few examples

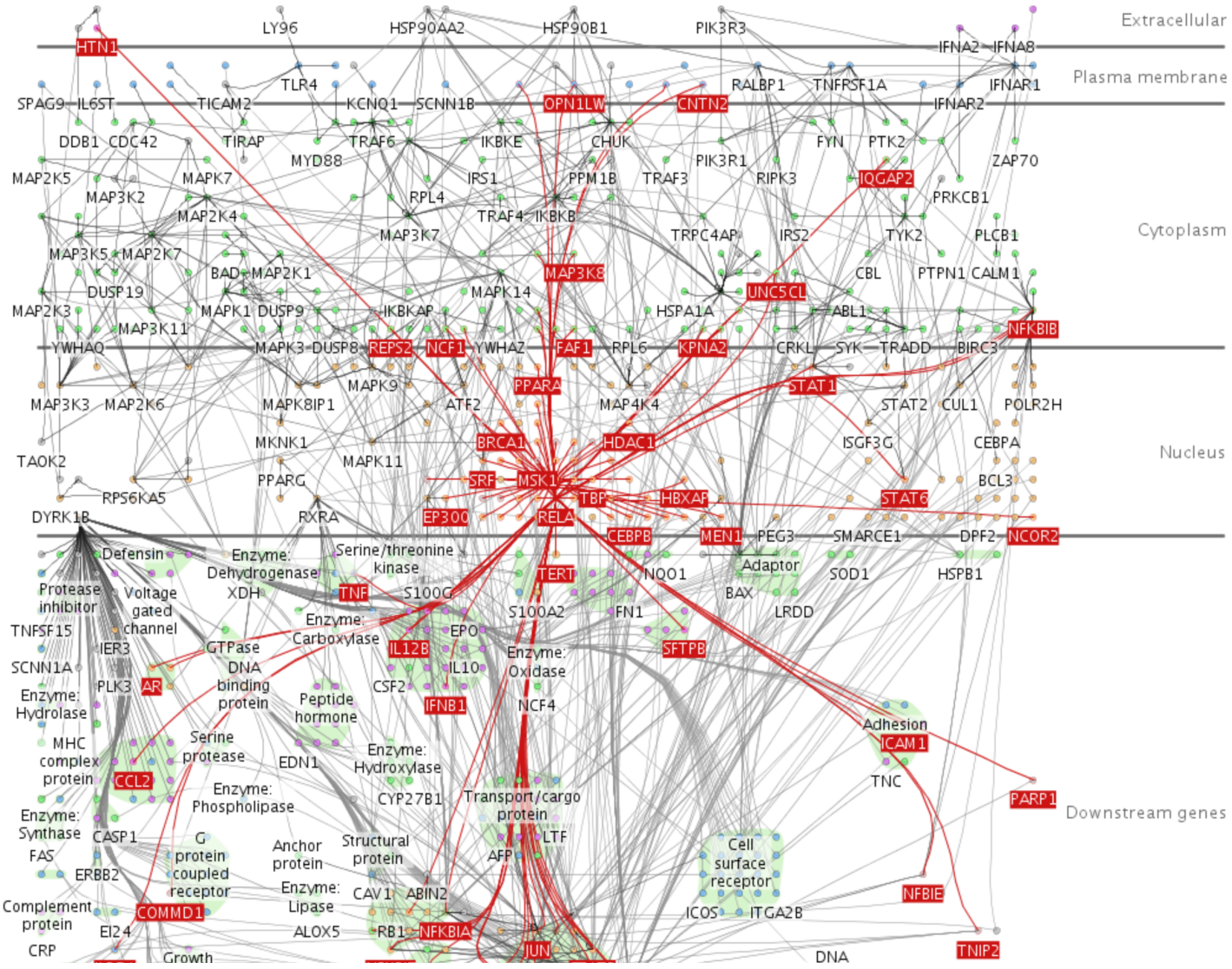
Food webs and networks

Metabolic networks,

Gene interaction or gene influence networks and graphs
Protein-Protein interaction networks

Brain pathway graphs - connectome


CHEBI:wortmannin
CHEBI:insulin
PFM:Prka Family
CHEBI:okadaic acid
HGNC:EGF
HGNC:MAP3K5
HGNC:ADRB1
NCM:pBS/p110 PI3Kinase Complex
PFH:MAPK JNK Family
GO:glucose import
NCM:Calcineurin Complex
SCHEM:Isoproterenol
HGNC:IGF1
HGNC:PTEN
HGNC:PRKCZ
NCH:pBS/p110 PI3Kinase Complex
PFH:Akt Family
CHEBI:resveratrol
PFH:AKT Family
HGNC:AKT1
SDIS:liver insulin receptor signaling pathway
CHEBI:hydrogen peroxide
SDIS:food intake
SDIS:muscle glucose import
GO:muscle contraction
HGNC:MAPK1
PFH:PRKC Family
SDIS:adipose glucose import
HGNC:TNF
SDIS:liver glucose import
SDIS:muscle insulin receptor signaling pathway
SCHEM:D-Glucose anhydrous
CHEBI:reactive oxygen species
SCHEM:serum leptin
HGNC:ADRB2
EBID:3650
HGNC:HNF1B
CHEBI:troglitazone
CHEBI:rosiglitazone
SCHEM:serum insulin
HGNC:IL6
NCH:AMP Activated Protein Kinase Complex
GO:insulin receptor signaling pathway
PFH:ADRB Family
HGNC:UCP2
GO:insulin secretion
CHEBI:lipid
HGNC:IRS1
SCHEM:serum glucose
HGNC:HIF1A
CHEBI:ceramide
HGNC:GCK
SCHEM:Thiazolidinedione
HGNC:SOCS3
PFH:IkappaB Kinase Family
GO:glucose transport
HGNC:HNF1A
HGNC:TGFA
CHEBI:nitric oxide
CHEBI:metformin
HGNC:YBX1
HGNC:IKBKB
HGNC:SOCS2
CHEBI:palmitic acid
HGNC:PARP1
SDIS:Diabetes
CHEBI:insulin
HGNC:PDX1
HGNC:PPARG
MESHPP:Insulin Resistance
biological process
protein abundance
chemical/metabolite abundance
catalytic activity
cell secretion
kinase activity
phosphatase activity
transcriptional activity
increases
decreases
directly increases


Extracellular
Plasma membrane
Cytoplasm
Nucleus
Downstream genes
HTN1
LY96
HSP90AA2
HSP90B1
PIK3R3
IFNA2
IFNA8
TLR4
RALBP1
TNFRSF1A
IFNAR1
SPAG9
IL6ST
TICAM2
KCNQ1
SCNN1B
OPN1LW
CNTN2
IFNAR2
DDB1
CDC42
TIRAP
TRAF6
IKBKE
CHUK
FYN
PTK2
MYD88
PIK3R1
ZAP70
MAP2K5
MAPK7
IRS1
PPM1B
TRAF3
RIPK3
IQGAP2
MAP3K2
RPL4
PRKCB1
MAP2K4
TRAF4
IKBKB
MAP3K7
TRPC4AP
IRS2
TYK2
PLCB1
MAP3K5
MAP2K7
BAD
MAP2K1
MAP3K8
CBL
PTPN1
CALM1
DUSP19
MAPK14
UNC5CL
MAP2K3
MAPK1
DUSP9
IKBKAP
HSPA1A
ABL1
MAP3K11
NFKBIB
YWHAQ
MAPK3
DUSP8
REPS2
NCF1
YWHAZ
FAF1
RPL6
KPNA2
CRKL
SYK
TRADD
BIRC3
MAPK9
PPARA
STAT1
MAP3K3
MAP2K6
MAPK8IP1
ATF2
MAP4K4
STAT2
CUL1
POLR2H
MKNK1
BRCA1
HDAC1
ISGF3G
CEBPA
TAOK2
MAPK11
SRF
MSK1
BCL3
PPARG
RPS6KA5
TBP
HBXAP
STAT6
DYRK1B
RXRA
EP300
RELA
CEBPB
MEN1
PEG3
SMARCE1
DPF2
NCOR2
Defensin
Enzyme: Dehydrogenase
Serine/threonine kinase
TERT
NQO1
Adaptor
SOD1
HSPB1
Protease inhibitor
Voltage gated channel
XDH
TNF
S100G
S100A2
FN1
BAX
LRDD
TNFSF15
Enzyme: Carboxylase
EPO
IER3
GTPase
IL12B
Enzyme: Oxidase
SFTPB
SCNN1A
DNA binding protein
IL10
PLK3
AR
CSF2
Enzyme: Hydrolase
Peptide hormone
IFNB1
NCF4
Adhesion
ICAM1
MHC complex protein
Serine protease
EDN1
Enzyme: Hydroxylase
TNC
CCL2
Enzyme: Phospholipase
CYP27B1
Transport/cargo protein
PARP1
Enzyme: Synthase
CASP1
G protein coupled receptor
Anchor protein
Structural protein
LTF
FAS
AFP
Cell surface receptor
ERBB2
Enzyme: Lipase
CAV1
ABIN2
NFBIE
Complement protein
COMMD1
ICOS
ITGA2B
EI24
ALOX5
RB1
NFKBIA
CRP
JUN
DNA
TNIP2
Growth

Biologically-Informative Features of Graphs and Networks

Simple features: Hubs, Cuts, Paths, Distances

High-density subgraphs: A non-trivial feature

A more complex feature of biological graphs that is thought to have significant biological importance is the variation in *density* in the graph, and the existence of *high-density* subgraphs.

Subgraphs and Density

Graph H has n nodes. The *density* of H, is the number of edges in H divided by $n(n-1)/2$, a number between 0 and 1.

A *high-density* subgraph H' of H is a subgraph of H, where the density of graph H' is above some large, fixed *threshold*, say 0.8.

A *clique* in graph $H = (V, E)$ consists of a subset V' of V (possibly all of V) where for *every* pair of nodes (u, v) in V', edge (u, v) is in E. The density of a clique is 1.

The Maximum Clique Problem: Our First ILP Formulation

Problem: Given an undirected graph G, find a maximum-size clique K in G.

The solution logic: If K is a clique, it is required that *if* a node i is chosen to be in K, *and* a node j is chosen to be in K, *then* (i, j) *must* be an edge in G.

That is equivalent to saying that if (i, j) is *not* an edge in G, then we *must not* choose *both* i and j to be in K. That is the logic that we will implement as linear inequalities.

The ILP variables and inequalities

The variables
One *binary* variable, $C(i)$, for each node i of G.

Variable $C(i)$ indicates whether or not node i will be included in a set called K^*. If $C(i)$ is set to 1 in the optimal ILP solution, we will put node i into K^*; and if $C(i)$ is set to 0, we will not. K^* will be a maximum-sized clique in G.

The 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.

$$C(i) + C(j) \leq 1 \tag{5}$$

The objective function

We use the objective function:

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

The objective function, along with the inequalities specified in (5), ensures that an optimal ILP solution will specify a *maximum-size* clique in G. Hence K^* will be one.

A concrete problem instance

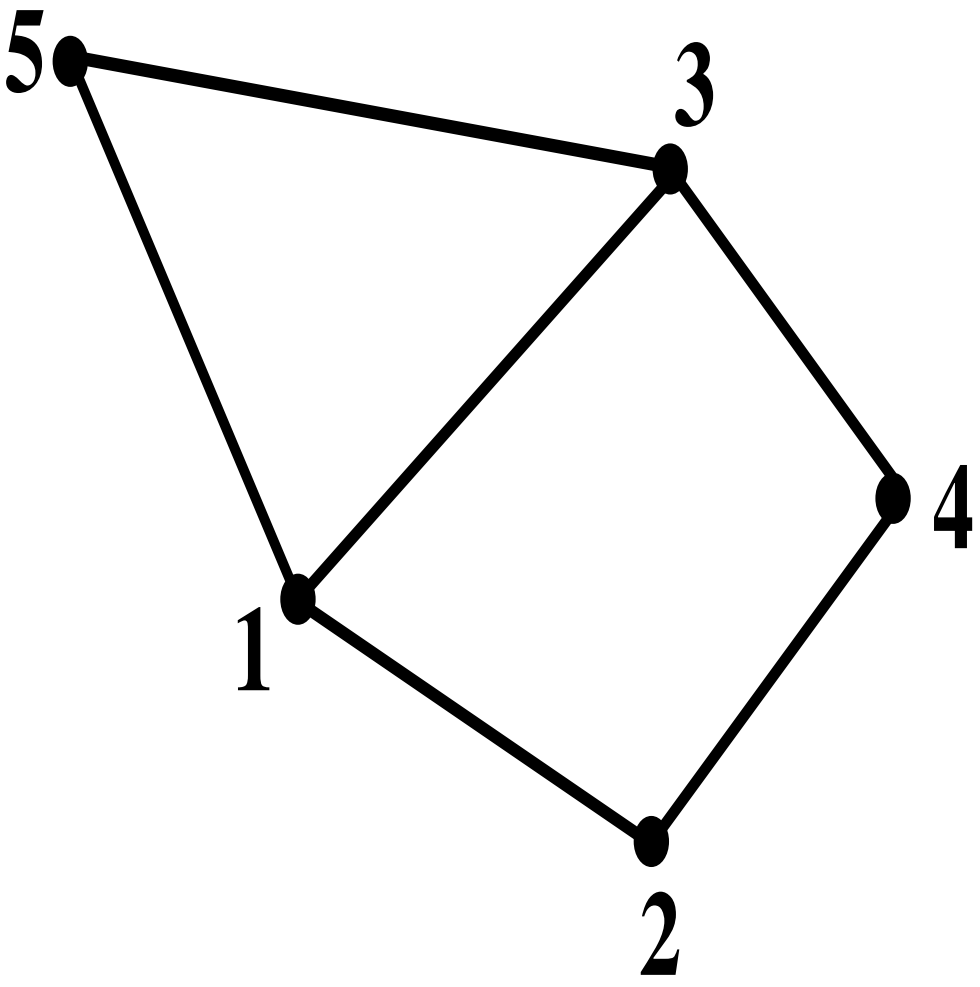


Figure : graph for max-clique ILP

The concrete ILP

The edges that are not in G are $(1,4),(2,3),(2,5)$ and $(4,5)$, the concrete ILP formulation for this instance of the maximum clique problem is:

$$\text{Maximize} \quad C(1)+C(2)+C(3)+C(4)+C(5)$$

Subject to

$$\begin{aligned} C(1)+C(4) &\leq 1 \\ C(2)+C(3) &\leq 1 \\ C(2)+C(5) &\leq 1 \\ C(4)+C(5) &\leq 1 \end{aligned} \tag{6}$$

where all variables are binary.

Formatting for Gurobi

Input that could be given to Gurobi for this small example is:

```
Maximize  C(1) + C(2) + C(3) + C(4) + C(5)

subject to
C(1) + C(4) <= 1
C(1) + C(4) <= 1
C(2) + C(3) <= 1
C(2) + C(5) <= 1
C(4) + C(5) <= 1

binary
C(1)
C(2)
C(3)
C(4)
C(5)
end
```

Solving the concrete ILP formulation

Save the above Gurobi-formatted ILP formulation in a file, say *clique.lp*.
The extension tells Gurobi the ILP formulation is in *LP format*.
To find an optimal solution to the concrete ILP formulation stored in *clique.lp*, I use a *terminal* window; move to the same directory that clique.lp is in; and issue the following command at the prompt:

```
gurobi_cl clique.lp
```

This tells Gurobi to read in file clique.lp and then (using default parameter settings for Gurobi) find values for the variables giving an optimal solution. Gurobi executes and outputs:

```
Gurobi Optimizer version 6.5.0 build v6.5.0rc1 (mac64)
Copyright (c) 2015, Gurobi Optimization, Inc.

Read LP format model from file clique.lp
Reading time = 0.00 seconds
(null): 4 rows, 5 columns, 8 nonzero
Optimize a model with 4 rows, 5 columns and 8 nonzeros
Coefficient statistics:
  Matrix range    [1e+00, 1e+00]
  Objective range [1e+00, 1e+00]
  Bounds range    [1e+00, 1e+00]
  RHS range       [1e+00, 1e+00]
Found heuristic solution: objective 2
Presolve removed 4 rows and 5 columns
Presolve time: 0.00s
Presolve: All rows and columns removed

Explored 0 nodes (0 simplex iterations) in 0.00 seconds
Thread count was 1 (of 8 available processors)
```

Seeing the full solution

If we want to see the values that Gurobi set for the variables in the optimal solution, we would execute Gurobi with the command

```
gurobi_cl resultfile=clique.sol clique.lp
```

The prefix 'resultfile=' is a Gurobi specification, but we make up the name of the file to write to.

```
# Objective value = 3
C(1) 1
C(2) 0
C(3) 1
C(4) 0
C(5) 1
```

Practicality

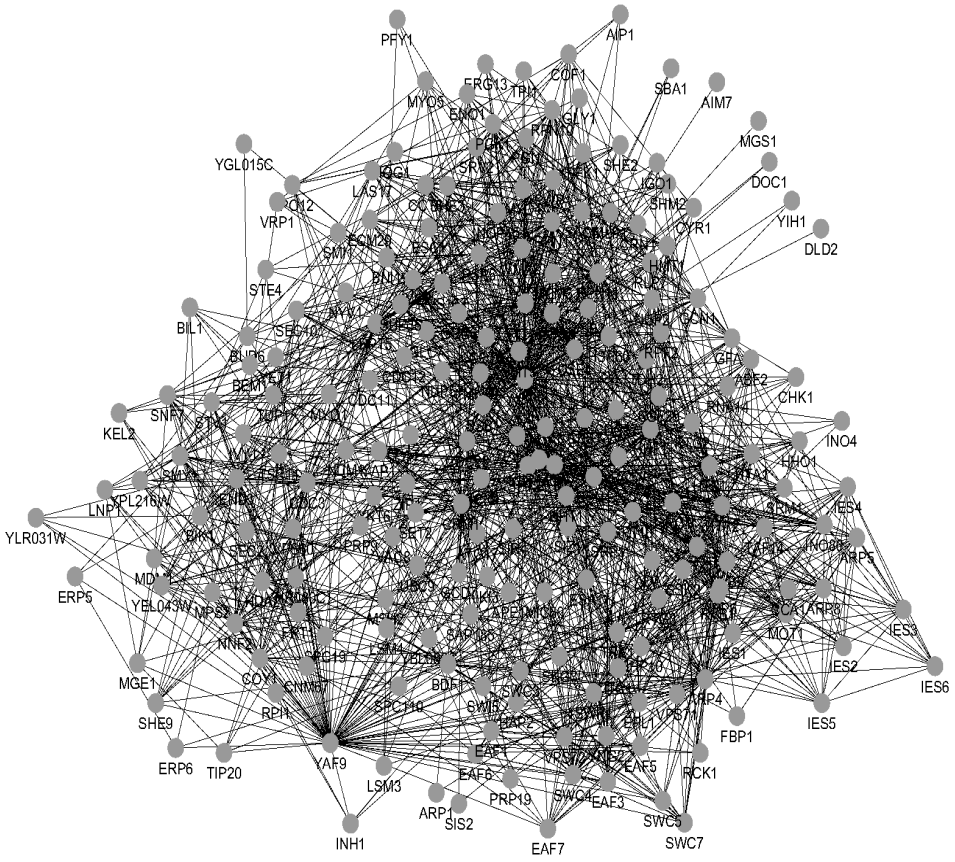
For randomly generated graphs with *one hundred* nodes, where each potential edge is in the graph with probability one-half, Gurobi 6.5 (on my Macbook Pro - i7) typically finds a maximum-size clique in under one second.

With 300 nodes, where each potential edge is in the graph with probability one quarter, Gurobi finds a maximum-sized clique in under ten seconds.

But for 500 nodes with the same edge density, the typical solution time increases to around 40 minutes. Whether that is a practical time depends on the user and the application.

Real biological graphs

To illustrate the practicality of maximum-clique finding in real biological networks, consider:



A "Hairball"

The figure shows part of the protein-protein interaction (PPI) graph[1] from a class of *yeast*. There are 208 nodes and 1776 edges, so the edge edge density is about 8.2%.

Gurobi solved the Maximum-Clique problem for this graph, finding a clique of size twelve, in about one-half of a second.

The Maximum-Clique Problem for randomly generated graphs with 208 nodes and edge-probability of 0.082 took 2.5 - 4 seconds to solve. The faster solutions for real biological networks is likely due to the greater asymmetry in real graphs compared to random graphs.

[1] Thanks to David Amar for producing this graphic.

Adding node weights

Suppose that $w(i)$ represents the weight given to node i. Then the ILP formulation for this problem is the same as for the Maximum Clique Problem, by changing the objective function from:

$$\sum_i C(i),$$

to

$$\sum_i w(i) \times C(i).$$

Later, we will see how to incorporate *edge* weights into the clique problem.

Bounds and Gurobi Progress Reporting

ILP solvers solve a concrete ILP formulation by alternately focusing on finding better *feasible solutions*, and by finding better *bounds* on the value of an optimal ILP solution.

In the case of a *maximization* problem, the solver creates a series of better feasible solutions, with value denoted *lb*, and a series of guaranteed *upper* bounds, denoted *ub*, on the value of an optimal solution.
Therefore, if *opt* denotes the optimal value for a problem instance, then at any point during the computation, it is guaranteed that $lb \leq \text{opt} \leq ub$, for the current values of *lb* and *ub*.

Progress reporting

```
    Nodes    |  Current Node    |     Objective Bounds      |     Work
 Expl Unexpl |Obj  Depth IntInf | Incumbent    BestBd   Gap |It/Node Time

     0     0   24.19001    0  150    9.0000   24.19001   169%     -    0s
H    0     0                       11.000000   24.19001   120%     -    0s
     0     0   23.85906    0  149   11.0000   23.85906   117%     -    0s
     0     0   23.71799    0  150   11.0000   23.71799   116%     -    0s
     0     0   23.71223    0  150   11.0000   23.71223   116%     -    0s
     0     0   23.66983    0  150   11.0000   23.66983   115%     -    0s
     0     0   23.65555    0  150   11.0000   23.65555   115%     -    0s
     0     0   23.65259    0  150   11.0000   23.65259   115%     -    0s
     0     0   23.65249    0  150   11.0000   23.65249   115%     -    0s
     0     0   23.61698    0  150   11.0000   23.61698   115%     -    0s
     0     0   23.61314    0  150   11.0000   23.61314   115%     -    0s
     0     0   23.61162    0  150   11.0000   23.61162   115%     -    0s
     0     0   23.61114    0  150   11.0000   23.61114   115%     -    0s
     0     2   23.61114    0  150   11.0000   23.61114   115%     -    0s
*  208   166               16       12.000000   22.88016  90.7% 80.3    1s
H 1031   575                       13.000000   19.57340  50.6% 65.9    2s
  1737   535   18.62161   22   81   13.0000   19.57340  50.6% 61.4    5s

Cutting planes:
  Clique: 38

Explored 6738 nodes (307332 simplex iterations) in 8.46 seconds
Thread count was 8 (of 8 available processors)

Optimal solution found (tolerance 1.00e-04)
Best objective 1.3000000000e+01, best bound 1.3000000000e+01, gap 0.0%
```

Part III. Near-Cliques in Biological Networks

We have developed an ILP formulation for the maximum-clique problem and discussed applications of it in biological networks. However, those were just two of the *wide* variety of problems and models in computational biology that involve finding large cliques, near cliques and *high-density subgraphs*.

Now, we extend the range of applications, with several *near-clique* problems, develop ILP formulations for them, and discuss some of the ways that these problems arise in computational biology. This discussion will also lead to our first ILP *idiom*, i.e., a formalized way to implement certain logical statements as integer linear inequalities.

Near Cliques

We are interested in finding a subset of nodes with a high *percentage* of the possible edges, but the requirement of having every possible edge is too severe.

There are many ways to formalize the concept of a *near clique*. We start with the simplest one:

> **The Maximum Near-Clique Problem:** *Find the largest subset of nodes K which would be a true clique if one edge was added between some pair of nodes in K.*

An ILP formulation to solve the Maximum Near-Clique Problem

We will modify the ILP for the maximum clique problem. In the max clique ILP the inequality

$$C(i) + C(j) \leq 1 \quad \text{if (i,j) is not an edge} \tag{7}$$

made it *impossible* to select both nodes i and j if (i, j) is not an edge.

Now, remove the inequalities that *strictly* prohibit this, and instead, use variables and inequalities that *record* that such a pair of nodes has been selected. Then we use an inequality that *limits* the number of such selected pairs.

Let $\overline{E}$ be the set containing every *pair* of nodes (i, j) in G where there is *no* edge between i and j.

For every pair (i, j) in $\overline{E}$, binary variable, $V(i, j)$ will record that both nodes i and j have been selected, although there is no edge between them. So, we replace the inequalities in (7) with:

$$C(i) + C(j) - V(i, j) \leq 1. \tag{8}$$

This says that *If both* $C(i)$ and $C(j)$ are set to 1, so that their sum is two, *then* variable $V(i, j)$ *must* be set to 1 in order to satisfy inequality (8).

Also, if $C(i)$ and $C(j)$ are both 0, inequality (8) reduces to $V(i, j) \geq -1$. If exactly one of $C(i)$ or $C(j)$ has value 1, then (8) reduces to $V(i, j) \geq 0$. These are trivially satisfied, so the inequalities do not have any bad side effects.

Hence, if we make those changes also add the inequality

$$\sum_{(i,j)\in\overline{E}} V(i,j) \leq 1,$$

then the set of selected nodes, K, will be a clique or a near-clique.

Summarizing, the *Maximum Near-Clique Problem* can be formulated as:

$$\text{Maximize } \sum_{i\in V} C(i)$$

subject to

$$\sum_{(i,j)\in\overline{E}} V(i,j) \leq 1, \tag{9}$$

and for each node pair $(i,j) \in \overline{E}$:

$$C(i) + C(j) - V(i,j) \leq 1, \tag{10}$$

where all variables are binary.

Note the asymmetry

The inequality $C(i) + C(j) - V(i,j) \leq 1$ ensures that $V(i,j)$ will be set to 1 *if both* $C(i)$ and $C(j)$ are set to 1.

But, it does *not* force $V(i,j)$ to be 0 when $C(i)$ and $C(j)$ are *not* both 1, and there is no inequality in the formulation that does that. This does not cause a problem, because the inequality

$$\sum_{(i,j)\in\overline{E}} V(i,j) \leq 1$$

limits the number of V variables that can be set to value one.

Related notions of a near clique

Suppose that $k > 1$ new edges can be added. Implementing that change in the ILP only requires that we change the right-hand side of inequality (9) from 1 to k.

We solved the Near-Clique problem for the *Yeast PPI* graph discussed earlier. The largest clique in that graph is of size 12.

Gurobi took ten minutes to determine that the largest clique remains of size 12, when only *one* new edge can be added; it took seven minutes to find a clique of size 13, when *two* new edges are allowed; and took 12 minutes to determine that the largest clique remains of size 13, when *three* new edges are allowed.

Another generalization

Allow up to one new edge *per node* to be added.

Implementing that change only requires that we replace the single inequality (9) with n inequalities, one for each node in G. For a node i, the inequality would be

$$\sum_{j:(i,j)\in\overline{E}} V(i,j) \leq 1.$$

An ILP for the Largest High-Density Subgraphs

> **The High-Density Subgraph Problem** *Given an undirected graph $G = (V, E)$, and a density threshold d between 0 and 1, find a subgraph, $G' = (V', E')$, in G with the maximum number of nodes, such that the density of G' is greater or equal to d.*

As before, we have binary variable $C(i)$ for each *node* i in G, indicating whether or not node i will be included in V'.

We also have a variable $P(i, j)$ for each *pair* of nodes (i, j) in G. This indicates that (i, j) is a *Potential* edge in E'. We also have a variable $E(i, j)$ for each *edge* $(i, j) \in G$. All of these variables are binary.

An ILP for the Largest High-Density Subgraphs

Variable $P(i,j)$ will be set to 1 in the optimal ILP solution *if* both $C(i)$ and $C(j)$ are set to value 1. This is implemented by:

$$C(i) + C(j) - P(i,j) \leq 1 \tag{11}$$

Note that

$$\sum_{i \in V, j \in V, i<j} P(i,j)$$

is an upper bound on the *maximum* number of edges that could possibly be in E', i.e., $n'(n'-1)/2$, where $n' = |V'|$. That fact that it is not exactly the maximum number, but only an upper bound, will not cause any trouble, due to the way it is used.

An ILP for the Largest High-Density Subgraphs

Variable $E(i,j)$ will be set to 1 *only if* both $C(i)$ and $C(j)$ are set to 1. This indicates that edge (i,j) will be in E'. This is implemented by:

$$2E(i,j) - C(i) - C(j) \leq 0 \quad \text{for (i,j) in E} \tag{12}$$

An ILP for the Largest High-Density Subgraphs

Next, we need an inequality that implements the requirement that the chosen nodes identify a subgraph with density threshold at least d. That is equivalent to the requirement that:

$$\frac{\sum_{i \in V, j \in V, i<j} [E(i,j)]}{\sum_{i \in V, j \in V, i<j} [P(i,j)]} \geq d \tag{13}$$

which is equivalent to:

$$\sum_{i \in V, j \in V, i<j} [E(i,j)] - d \times \sum_{i \in V, j \in V, i<j} [P(i,j)] \geq 0, \tag{14}$$

where d is the density threshold, between 0 and 1, specified by the user.

Finally, the objective function for the formulation is

$$\text{Maximize} \sum_i C(i).$$

Practicality

The meaning of "practical" depends on the full context of the problem instance. If it takes years to accumulate the data (which is the case in some PPI data) then an ILP solution time of several days is a practical computation.

We tested the ILP formulations on random graphs, and solved them with Gurobi, for a range of values for n, p and d.

The formulations take longer to solve than do the formulations for the maximum clique problems with the same n and p. Still the range of n and p where the approach is successful illustrates the power of the ILP approach compared to brute-force methods.

n	p	d	seconds for 6.5	seconds for 7.0
25	0.4	0.8	0.6	0.58
25	0.6	0.9	0.66	0.82
30	0.4	0.9	1.13	0.77
35	0.5	0.8	9.03	8.73
50	0.4	0.8	166	102
50	0.2	0.5	250	268

However, when the ILP formulation was applied to a much larger problem, the 208 node graph for the *yeast* PPI network discussed earlier, where the maximum-size clique of size 12 was found in under a second, the ILP formulation for the maximum density problem, with $d = 0.8$, did not complete for three hours and was stopped.

A more complete test was run on that data with $d = 0.5$. Gurobi was allowed to run for a full day before it was stopped. At that point, it had found a subgraph of 36 nodes with a density of 0.5 or greater, and a *guarantee* that no high-density subgraph (with $d = 0.5$) of size 63 exists in the PPI network. Moreover, the execution found a subgraph of size 30 (with $d = 0.5$) in the first 21 minutes of execution, and found a subgraph of size 35 after about seven hours of execution.

Part IV: Our first ILP Idioms

There are certain *logical constructs* that come up frequently in mathematical modeling, and general linear integer inequalities have been developed to implement them. We use the term "idiom" to refer to such logical constructs and their ILP implementations.

An "If-Then" idiom for binary variables
Overwhelmingly, the most important ILP idiom that arises in computational biology is the *If-Then* idiom. For example, inequality (8) expresses the logical construct that *if* $C(i) + C(j)$ is equal to 2, *then* variable $V(i, j)$ *mus*t be *set* equal to 1.

This is a simple instance of the *If-Then* idiom for *binary* variables. More general versions will be discussed as the tutorial progresses.

An "Only-If" ILP idiom for binary variables

Inequality (8) sets $V(i,j)$ to 1 *if* $C(i) + C(j)$ equals 2, but it does not prevent $V(i,j)$ from being set to 1 *even if* $C(i) + C(j)$ is *less than* 2.

In some formulations, we need to enforce the *converse* of an *If-Then* construct. In the case of inequality (8), the converse construct is:

> $V(i,j)$ *is allowed to be set to 1 only if* $C(i) + C(j)$ *is equal to 2.*

To implement this Only-If idiom, we might try the inequality:

$$C(i) + C(j) - V(i,j) \geq 1 \tag{15}$$

This looks promising, but is not correct. It has a bad *side-effect*.

When $V(i,j)$ has value 0, inequality (15) reduces to $C(i) + C(j) \geq 1$, which is bad.

A correct "Only-If" inequality

To implement the *Only-If* idiom without bad side effects:

$$2V(i,j) - C(i) - C(j) \leq 0 \tag{16}$$

When $V(i,j)$ has value 1, both $C(i)$ and $C(j)$ must be set to 1.
When $V(i,j)$ has value 0, inequality (16) reduces to
$-C(i) - C(j) \leq 0$, which is always satisfied.

More general "If-Then" and "Only-If" idioms for binary variables

Suppose L is an integer linear function of binary variables. A more general version of the *If-Then* idiom is:

> *If the binary ILP variables are set so that the integer linear function L has value greater or equal to a strictly positive value, b, then a binary variable, z, must be set to 1. Variable z is called an "indicator" variable.*

For example, consider the statement::

> If $5x_1 + 7x_2 - 8x_3 + 2x_4 \geq 2$, Then z must be set to 1

All of the variables $x_1, ..., x_4, z$ are binary.

Implementing this general "If-Then" idiom

Note that the variables in L are binary, so there is an *upper limit* on the largest value that L can attain. Let M denote the largest value that the linear function L can attain. Then, the following abstract inequality implements the general *If-Then* idiom for binary variables:

$$L - (M \times z) \leq b - 1 \tag{17}$$

A more general "Only-If" idiom for binary variables

$z = 1$ *only if the value of* L *is greater or equal to* b*.*

To implement that idiom, we might first try:

$$L + z \geq b + 1.$$

When $z = 1$, the inequality becomes $L \geq b$, which can be satisfied *only if* the value of L is greater or equal to b, so this looks promising. However, the inequality has a bad side-effect because when $z = 0$, the inequality becomes $L \geq b + 1$.

To get a correct implementation, let s be the *smallest* value that function L can achieve and set $m = s - b$ (i.e., $s = m + b$). Then, we can correctly implement the *Only-If* idiom with the inequality:

$$L + m \times z \geq m + b \tag{18}$$

An immediate extension from binary variables to bounded variables

Above, we assumed that the all the variables in L were binary. That was used to establish upper and lower bounds, M and s, on the values that L can attain, and such bounds were all that were needed for the the validity of the idioms.

If the variables in L are *not* necessarily binary, but are *bounded* (i.e., for each variable in L, there are *known* upper and lower bounds on the values that the variable can attain), then the values that L can attain are also bounded, both above and below.

Exploiting the idioms

The *If-Then* and *Only-If* idioms can be used to build additional idioms.

The NOT-AND (NAND) idiom for inequalities Let L_1 and L_2 be linear functions whose variables are bounded, and consider the linear inequalities, $L_1 \geq b_1$ and $L_2 \geq b_2$. Suppose we require that *at most one* of the two linear inequalities is satisfied. This is the NOT-AND idiom for *inequalities*.

The NAND idiom can be implemented by using the *If-Then* idiom twice: once, so that *if* $L_1 \geq b_1$, *then* variable z_1 is set to value 1; and once so that *if* $L_2 \geq b_2$, *then* variable z_2 is set to value 1. Next, we add the inequality:

$$z_1 + z_2 \leq 1 \tag{19}$$

The OR idiom for inequalities

Suppose we require that *at least* one of the inequalities is satisfied. This is the OR idiom for *inequalities*.[2]
We use the *Only-If* idiom twice: once, so that z_1 is set to 1 *only if* $L_1 \geq b_1$; and once so that z_2 is set to 1 *only if* $L_2 \geq b_2$. Then, we replace inequality (19) with:

$$z_1 + z_2 \geq 1 \tag{20}$$

If neither L_1 nor L_2 can ever take on a negative value, then the *OR* idiom for inequalities can be implemented more simply as:

$$\begin{aligned} L_1 &\geq (1 - z)b_1 \\ L_2 &\geq z \times b_2, \end{aligned} \tag{21}$$

where z is a binary variable.

[2]The OR idiom for *binary variables* is simpler: $X + Y \geq 1$.

An XOR idiom for inequalities

To require that *exactly* one of the inequalities is satisfied, we use the inequalities for the *If-Then* idioms for both L_1 and L_2; and the inequalities for the *Only-If* idioms for both L_1 and L_2.

So, z_1 will be set to 1 if and only if $L_1 \geq b_1$; and, z_2 will be set to 1 if and only if $L_2 \geq b_2$.

Then, we replace inequality (19) with:

$$z_1 + z_2 = 1 \tag{22}$$

This idiom is called the *Exclusive-Or (XOR)* idiom for *inequalities*.[3]

[3] The XOR idiom for two *binary variables*, X and Y, is simpler: $X + Y = 1$.

An IMPLIED-SATISFACTION idiom for inequalities

We can also express the construct:

$$\text{if } L_1 \geq b_1 \text{ then } L_2 \geq b_2,$$

by using the *If-Then* idiom for the first inequality, and the *Only-If* idiom for the second inequality, and then replacing inequality (19) with

$$z_1 \leq z_2 \tag{23}$$

which forces z_2 to have value 1 if z_1 has value 1.

A NOT-EQUAL idiom

In many ILP formulations, it is natural to *require* that two *integer* variables (whose values can be larger than 1) always take on *different* values. This is easily implemented using the *OR* idiom, as follows. If X and Y are integer variables, then the inequality:

$$(X - Y \geq 1) \; OR \; (Y - X \geq 1) \tag{24}$$

is satisfied *if and only if* the values of X and Y are different.

A NOT-EQUAL idiom for binary variables

The situation is simpler when X and Y are *binary* variables; then the NOT-EQUAL idiom is trivially implemented as $X + Y = 1$. Note that this *forces* the binary variables X and Y to have different values.

However, sometimes we only want to *test* if two binary variables have different values. We want the logical construct that *if* binary variable X is NOT-EQUAL to binary variable Y, *then* binary variable z is set to 1. This is implemented by:

$$\begin{aligned} z &\geq X - Y \\ z &\geq Y - X. \end{aligned} \tag{25}$$

NOT-EQUAL idioms with linear functions

More generally, suppose Z_1 and Z_2 are *linear functions* of integer variables whose values are *bounded* from both above and below, and can only have integer values. Then the functions $Z_1 - Z_2$ and $Z_2 - Z_1$ have bounded values that are always integers.

In that case, we can express the NOT-EQUAL idiom, $Z_1 \neq Z_2$ as:

$$(Z_1 - Z_2 \geq 1) \; OR \; (Z_2 - Z_1 \geq 1). \tag{26}$$

Of course, when creating a concrete ILP formulation, the OR idiom must be implemented with linear inequalities, as explained earlier.

A special case

A particularly simple case is that Z_1 and Z_2 are just integer *variables* whose values are bounded between 1 and n.

Then, starting with (26), and expanding the OR idiom, we can implement the requirement that $Z_1 \neq Z_2$ with the inequalities:

$$\begin{aligned} Z_1 - Z_2 - n \times (z - 1) &\geq 1 \\ Z_2 - Z_1 + n \times z &\geq 1 \end{aligned} \tag{27}$$

where z is a *binary* variable.

The Key to the Idioms

The key to all of these idioms for inequalities is that they first implement the logic that when a specific inequality is satisfied an *indicator variable*, such as z, z_1, z_2, is set to value 1.

Or, conversely, they implement the logic that when an indicator variable is set to 1, a specific inequality must be satisfied; or both.

Then, using an individual indicator variable for each inequality, we add constraints on the indicator variables to implement relations between the inequalities. Those relations are implemented by idioms for variables.

Part V. The RNA Folding Problem

The **RNA Folding Problem** is to predict the *secondary structure* of an RNA molecule, given only its nucleotide sequence. This important, classic problem in computational biology is often solved with variants of *dynamic programming* which have been highly-refined and engineered in several widely-used computer programs.

But here, we show how *integer linear programming* can be used to obtain the same results, with much less effort on the part of the developer or programmer, and can also be extended to model more complex versions of the folding problem, in ways that are difficult to model with dynamic programming.

We start with an ILP formulation for a simplified version of the RNA problem, and then extend the biological model and the ILP formulation to incorporate more realistic biological features of the problem.

A Crude First Model of RNA Folding

We let S denote a string of n characters made up of the RNA alphabet $\{A, C, U, G\}$. For example, $S = ACGUGCCACGAU$.

A *pairing* is set of *disjoint* pairs of characters in S. A character can be in at most one pair. Note that some characters might not be in any pair in a pairing.

A pair is called *complementary* if the two characters in the pair are $\{A,U\}$ or $\{C,G\}$. In our first model of RNA folding, we require that all pairs be complementary.

If we draw the RNA string S as a circular string, we define a *nested pairing* (alternately called *non-crossing*) as a pairing of *complementary* nucleotides, where each pair in the pairing is connected by a line *inside* the circle, and where *none* of the lines cross each other.

A nested pairing of complementary nucleotides

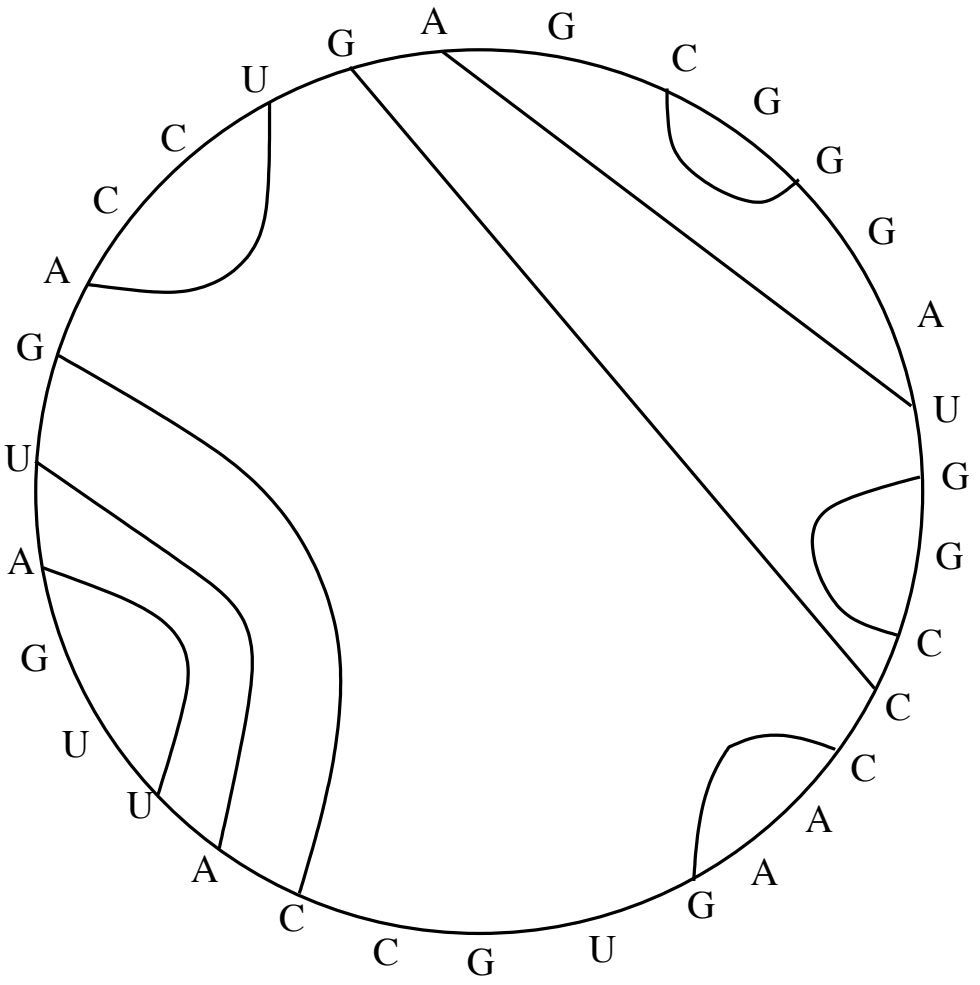


Figure : The lines show a nested pairing, not necessarily the largest.

Fold stability

It is generally asserted that *as a first approximation*, the fold of an RNA molecule *corresponds* to a nested pairing that is the most *stable*.

In the simple model, we measure the stability of a nested pairing by the *number* of matched pairs it has. So, the most stable nested pairing is the one with the *largest* number of matched pairs. This leads to the following computational problem:

> **The Simple RNA Folding Problem** *Given the nucleotide sequence S of a RNA molecule, find a nested pairing that pairs the maximum number of nucleotides, compared to any other nested pairing.*

Formulating and Solving the Simple RNA Folding Problem via ILP

We create one binary ILP variable, called $P(i, j)$, for each pair (i, j) of positions in S, where $i < j$. The value of $P(i, j)$ in a solution of the ILP will indicate whether or not the nucleotide in position i of S will be paired with the nucleotide in position j of S: value 1 if 'yes'; value 0 if 'no'.

If the ordered pair of nucleotides in any positions i and j are *not* (A, U) or (U, A) or (C, G) or (G, C), then we will create and include the equality:

$$P(i, j) = 0,$$

to disallow the pairing of the nucleotides in positions i and j.

Each site in at most one pair

Next, we implement the requirement that each nucleotide can be paired to *at most* one other nucleotide.

$$\text{for each j} \quad \sum_{k>j} P(j,k) + \sum_{k<j} P(k,j) \leq 1. \tag{28}$$

Note that this is an *inequality*, not an *equality*, meaning that it is permissible for a position j to *not* be in any pair.

Nesting required

To implement the requirement that the pairing be *nested*, the key is to note that a pairing that is *not* nested, must contain matched pairs (i, j) and (i', j') where $i < i' < j < j'$.
So, we use: For every choice of four positions $i < i' < j < j'$,

$$P(i, j) + P(i', j') \leq 1 \tag{29}$$

We can think of this as instance of an ILP idiom: the *NAND* ("Not And") of two *variables*. This is a particularly simple idiom, because it only involves relations between variables, rather than between inequalities.

The objective function

Finally, the objective function for the ILP is:

$$\text{Maximize} \sum_{i<j} P(i,j)$$

which says that we want to set as many P variables as possible to the value 1.

A Toy Example

$S = ACUGU$. Then the ILP is:
Maximize$P(1,2) + P(1,3) + P(1,4) + P(1,5) + P(2,3) +$
$P(2,4) + P(2,5) + P(3,4) + P(3,5) + P(4,5)$
s.t.

```
P(1,2) = 0
P(1,4) = 0
P(2,3) = 0
P(2,5) = 0
P(3,4) = 0
P(3,5) = 0
P(4,5) = 0

P(1,2) + P(1,3) + P(1,4) + P(1,5) <= 1
P(1,2) + P(2,3) + P(2,4) + P(2,5) <= 1
P(1,3) + P(2,3) + P(3,4) + P(3,5) <= 1
P(1,4) + P(2,4) + P(3,4) + P(4,5) <= 1
P(1,5) + P(2,5) + P(3,5) + P(4,5) <= 1
```

Continuing

```
P(1,3) + P(2,4) <= 1
P(1,3) + P(2,5) <= 1
P(1,4) + P(2,5) <= 1
P(1,4) + P(3,5) <= 1
P(2,4) + P(3,5) <= 1
```

Simple Biological Enhancements

- Minimum distance constraint between paired positions, to avoid impossible bends. Trivial to incorporate this constraint.
- Differential binding strengths.
  The binding strength of a $\{C, G\}$ pair is larger than the binding strength of an $\{A, U\}$ pair, since a $\{C, G\}$ pair has three hydrogen bonds, while an $\{A, U\}$ pair only has two bonds. So, to find the most *stable* nested pairing, we need a *maximum weight* nested pairing. Trivial to incorporate into the objective function.
- Allowing non-complementary matched pairs.

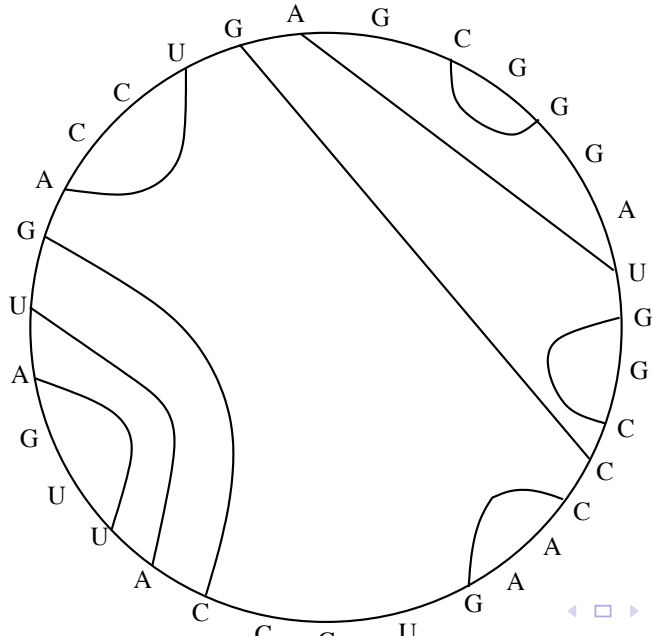
  In some models of RNA folding, certain *non-complementary* pairs of characters are allowed to form matching pairs, as long as appropriate weights, or multipliers, are used in the objective function for each allowed pair. The most commonly allowed non-complementary pair is $\{G, U\}$.

More Complex Biological Enhancements

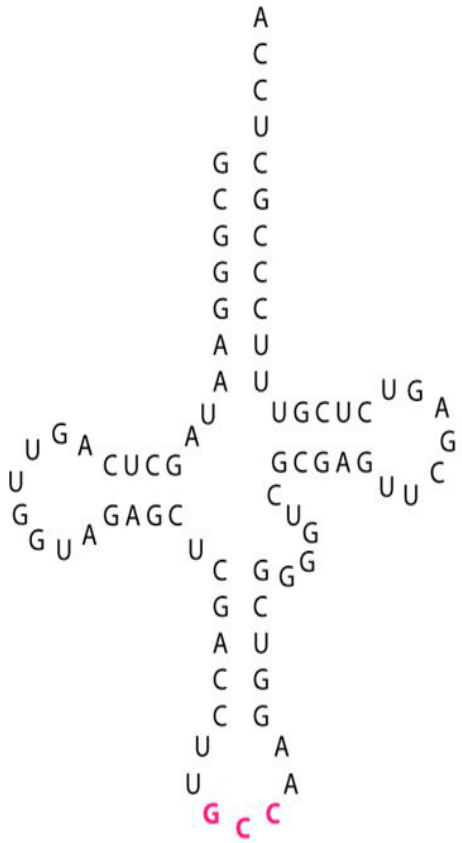
Base Stacking - stems - helices

A *matched* pair (i, j) in a nested pairing is called a *stacked pair* if either $(i+1, j-1)$ or $(i-1, j+1)$ is also a matched pair in the nested pairing.

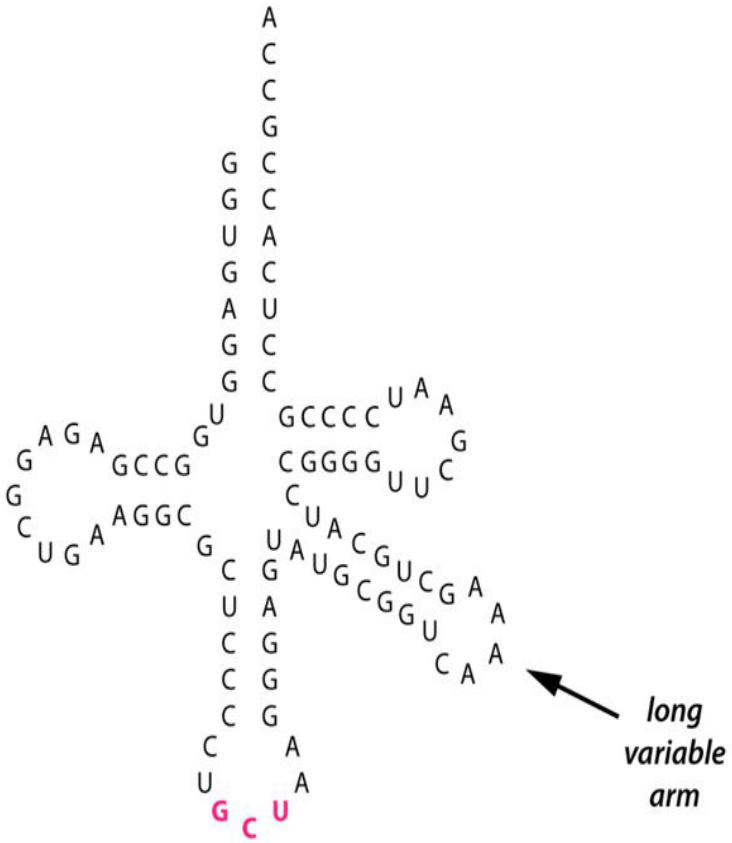
A *stack* in a nested pairing consists of a *consecutive* run of two or more stacked pairs. If (i, j) and $(i+1, j-1)$ are stacked pairs, the four positions $(i < i+1 < j-1 < j)$ is called a *stacked quartet*.



Stacks are particularly evident in the folding of *transfer RNA (tRNA)*, in a distinctive secondary structure called a *cloverleaf*.



tRNA^{Gly} (class I tRNA)



tRNA^{Ser} (class II tRNA)

Stacks and stability

Stacking contributes significantly to the stability of an RNA fold. So a more realistic ILP formulation for RNA folding, must *encourage* paired nucleotides to be organized into (long-ish) stacks as much as possible.

As a simple first step, we will extend the objective function of the ILP by including a count of the *number* of stacked quartets in the nested pairing. Hence, we need the ILP formulation to calculate that number, based on the pairing variables $P(i, j)$.

Counting quartets

We create the binary ILP variable $Q(i,j)$ to indicate whether the pair (i,j) is the *first* pair in a stacked quartet. We have, for each i,j, where $j > i$:

$$P(i,j) + P(i+1,j-1) - Q(i,j) \leq 1 \tag{30}$$

$$2Q(i,j) - P(i,j) - P(i+1,j-1) \leq 0 \tag{31}$$

The first inequality enforces the condition that *if both* (i,j) and $(i+1,j-1)$ are in the nested pairing *then* the value of variable $Q(i,j)$ *must* be set to 1.

The second inequality enforces the converse condition, that $Q(i,j)$ can be set to 1 *only if* $P(i,j)$ and $P(i+1,j-1)$ are both set to 1.

The two inequalities together ensure that $Q(i,j)$ will be set to 1, *if and only if* both (i,j) and $(i+1,j-1)$ are in the nested pairing, so $(i,i+1,j-1,j)$ is a stacked quartet, and i is the smallest position in the quartet.

Then, to incorporate a count of the number of stacked quartets in the fold, we change the objective function from

$$\text{Maximize} \sum_{i<j} P(i,j)$$

to

$$\text{Maximize} \sum_{i<j} [P(i,j) + Q(i,j)].$$

Weighting stacked quartets in a nested pairing

The next, and perhaps the most important, extension of the chemical model is to incorporate *weights* into the objective function for each stacked quartet in a stack.
Then the objective function is given as:

$$\text{Maximize} \sum_{i=1}^{i=n} [W(i,j) \times Q(i,j)],$$

where $W(i,j)$ is a positive constant that depends on which four nucleotides are in the stacked quartet $(i, i+1, j, j-1)$.

Quartet weights

Extensive chemical studies have been done to determine good weights for stacked quartets, based on the specific nucleotides that the stacked quartet contains.

The following table shows the weights for stacked quartets used in a program called Fold-Align:

	A U	C G	G C	U A
A U	9	21	24	13
C G	22	33	34	24
G C	21	24	33	21

More elaborate models of RNA folding

The two most important features of RNA folding are *complementary pairing*, and *base stacking*. Those two features, along with appropriate weights for matched pairs and stacked quartets, were adequate to explain both the central ideas in RNA fold prediction, and the formulation of ILPs for RNA fold prediction.

However, many additional features and refinements of RNA folds have been incorporated into fold prediction methods. In one software package for RNA folding, there are close to *two hundred* parameter choices that the user can specify to guide the folding algorithm.
In some models of RNA folding, the weight given to stacked quartet depends both on the specific nucleotides in the quartet, and on *where* the stacked quartet is in a stack. The main distinction is whether the stacked quartet is the *first* quartet, the *last* quartet, or a *middle* quartet in a stack.

Hence, we need to extend the ILP formulation to recognize where a stacked quartet appears in a stack. Let $F(i,j)$ be an ILP variable that will be set to 1 if and only if the stacked quartet $(i, i+1, j-1, j)$ is the *first* stacked quartet in a stack. We use:

$$Q(i,j) - Q(i-1,j+1) - F(i,j) \leq 0$$

and

$$2F(i,j) - Q(i,j) + Q(i-1,j+1) \leq 1.$$

Exercise: Crossing matched pairs

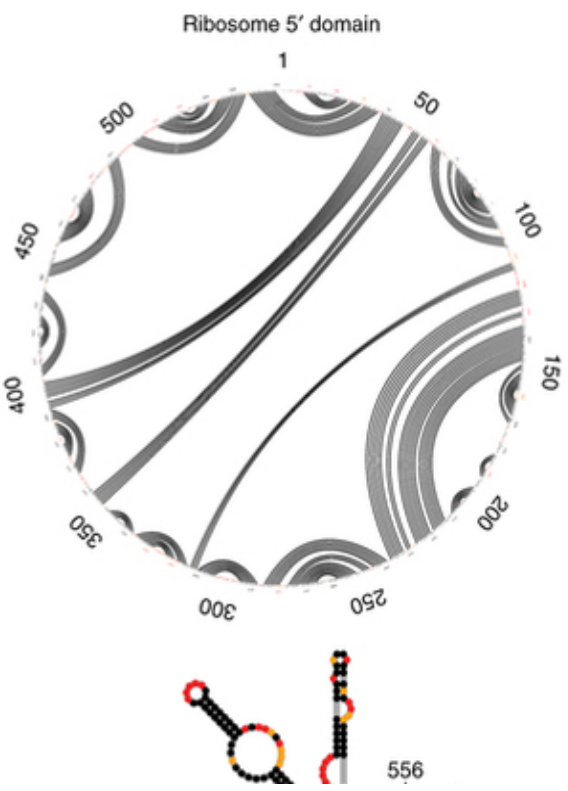
Up until now, we have required that pairings be *nested, non-crossing*, since they are thought to model the secondary structure of most tRNA molecules.
However, a limited number of crossing matched pairs are sometimes observed in RNA secondary structure, particularly for RNA molecules that are not tRNA molecules.

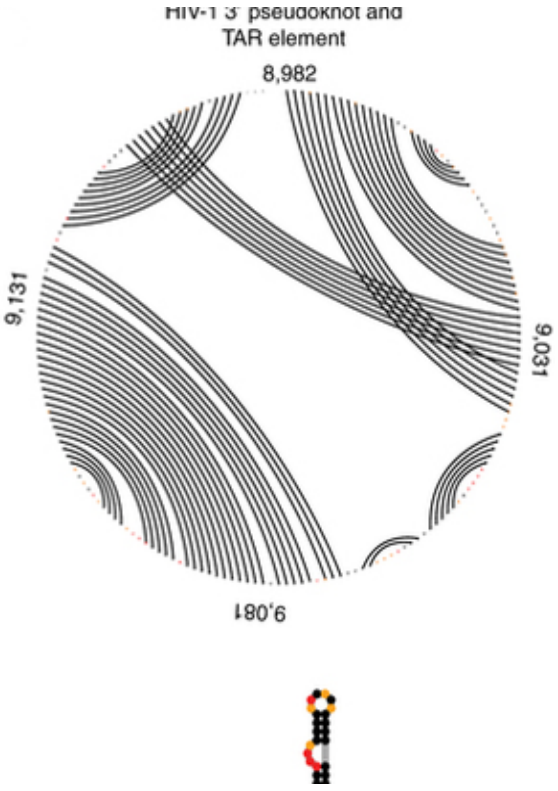
Suppose we now change the definition of a pairing to allow *some* crossing matched pairs.
Let $C(i, i', j, j')$ be an ILP variable that is set to value 1 if and only if (i, j) and (i', j') are matched pairs that cross.

Develop ILP inequalities to properly implement the definition of $C(i, i', j, j')$. Once the C variables are implemented, we we can add an inequality that limits the number of crossing matched pairs to a fixed number, or to a fixed percentage of the number of matched pairs in the pairing, etc.

Pseudo-Knots

One of the most important features (we have not discussed) of some RNA folds is called a *pseudo-knot*. Pseudo-knots generalize crossing pairs. If two sets of crossing matched pairs are organized into two stacks, they form a *single* pseudo-knot, and should be considered a single feature.





Exercise

Let $PS(i, i')$, for $i < i'$, be a binary ILP variable that will be set to 1 if and only if there is a pseudo-knot whose two stacks begin at positions i and i' respectively.

Develop ILP inequalities that correctly implement the definition of the variables $PS(i, i')$, for each $i < i'$. Then suggest and develop inequalities that allow a few pseudo-knots, or balance the inclusion of pseudo-knots with an increase in the number of stacked pairs.

Ending Comments

- ILP formulations benefit both from faster computers, and from faster solvers. Clever algorithms only benefit from the former.
- "What" is easier than "How". An ILP formulation is often a statement of what we *want* the optimal to look like. It rarely describes *how* to obtain it. We express the *what* in terms of a Huge number of inequalities with a Huge number of variables, and then let the ILP solver do the heavy lifting. At first, this seems unlikely to be practical (e.g., using millions of inequalities and tens of thousands of variables to encode a small problem instance), but it often is.
- How ILPs for Computational Biology differ from ILPs for traditional applications: Reductions!
- The virtue of NP-hardness: Expressibility!

- Why ILP and not some other NP-hard problem? In theory, any NP-hard problem will do. In practice, there is an ILP industry which invests, maintains, and supports. There is no similar industry for any other NP-hard problem. SAT-solvers might be competitive for decision problems, but they don't optimize naturally, and are very bulky even decision problems if the data is *weighted*.