Combinatorial Algorithms Used Inside a MIP Solver
A linear program (LP) is defined as

\[
\begin{align*}
\text{min} & \quad c^T x \\
\text{s.t.} & \quad Ax \leq b \\
& \quad x \in \mathbb{R}^n
\end{align*}
\]

A mixed integer program (MIP) is defined as

\[
\begin{align*}
\text{min} & \quad c^T x \\
\text{s.t.} & \quad Ax \leq b \\
& \quad x \in \mathbb{R}^n \\
& \quad x_j \in \mathbb{Z} \quad \text{for all } j \in I
\end{align*}
\]
Applications of Mixed Integer Programming

- Accounting
- Advertising
- Agriculture
- Airlines
- ATM provisioning
- Compilers
- Defense
- Electrical power
- Energy
- Finance
- Food service
- Forestry
- Gas distribution
- Government
- Internet applications
- Logistics/supply chain
- Medical
- Mining National research labs
- Online dating
- Portfolio management
- Railways
- Recycling
- Revenue management
- Semiconductor
- Shipping
- Social networking
- Sports betting
- Sports scheduling
- Statistics
- Steel Manufacturing
- Telecommunications
- Transportation
- Utilities
- Workforce scheduling
- ...
\( P \) vs \( NP \)

- **Problem class \( P \):**
  - Problem instance is solvable in worst-case runtime that is polynomial in input size
  - Examples:
    - Sorting
    - Shortest path
    - Maximum weighted matching
    - Linear program

- **Problem class \( NP \):**
  - Solution for given problem instance can be verified in polynomial time w.r.t. instance size
  - Obviously, \( P \subseteq NP \)

- **Problem class \( NP \)-complete:**
  - \( P \in NP \) is \( NP \)-complete if every problem in \( NP \) can be transformed into \( P \) using a polynomial transformation
  - Examples:
    - Satisfiability problem (SAT)
    - Knapsack
    - Traveling salesman problem
    - Maximum weighted clique
    - Integer program
• Theory says:
  • Linear programming is easy
    • Interior point algorithm has polynomial worst-case runtime
  • Integer programming is hard
    • Branch-and-cut has exponential worst-case runtime
      • exponential in number of integer variables
• Let’s look at problem sizes and runtime for real-world problem instances
  • LP test set has 2397 instances
  • MIP test set has 7030 instances
  • Gurobi 9.5.0
  • Intel Xeon CPU E3-1240 v3 @ 3.40GHz
  • 4 cores, 8 hyper-threads
  • 32 GB RAM
  • Time limit of 10,000 seconds
Linear Programming
Full test set

Smallest LP that Gurobi cannot solve in 10k sec
Original model:
- 345,684 constraints
- 101,564 variables
- 692,411 non-zeros

Presolved model:
- 344,505 constraints
- 100,383 variables
- 690,414 non-zeros

Interior point linear system:
- ~26,580,000 $AA^T$ non-zeros
- ~3,818,000,000 factor non-zeros

Solving statistics:
- hit time limit after 19 interior point iterations

Biggest LP that Gurobi can solve in 32GB RAM
Original model:
- 3,355,357 constraints
- 1,419,480 variables
- 316,017,220 non-zeros

Presolved model:
- 344,505 constraints
- 100,383 variables
- 690,414 non-zeros

Interior point linear system:
- ~13,430,000 $AA^T$ non-zeros
- ~69,620,000 factor non-zeros

Solving statistics:
- 102 interior point iterations
- 193.9 seconds
Linear Programming
Full test set

Solving time vs. Number of non-zeros
Linear Programming
Models with up to 100 million non-zeros

very roughly looks like a linear runtime behavior
Mixed Integer Programming
Full test set

Smallest MIP that Gurobi cannot solve in 10k sec

Original model:
- 74 constraints
- 56 variables (28 general integers)
- 168 non-zeros

Presolved model:
- 14 constraints
- 20 variables (20 general integers)
- 60 non-zeros

Solving statistics:
- hit time limit after 140,132,462 search nodes
- final MIP gap is 0.09%

Biggest MIP that Gurobi can solve

Original model:
- 5,088,000 constraints
- 3,379,700 variables (all binary)
- 328,860,900 non-zeros

Presolved model:
- 0 constraints
- 0 variables
- 0 non-zeros

Solving statistics:
- solved by presolve
- 69.8 seconds
Mixed Integer Programming
Models with up to 100 million non-zeros
Mixed Integer Programming
Models with up to 1 million non-zeros

No obvious relation between size (# non-zeros) and solve time
Mixed Integer Programming
Full test set

Smallest MIP that Gurobi cannot solve in 10k sec

Original model:
• 19,299 constraints
• 1,012,110 variables (17 binaries)
• 4,419,580 non-zeros

Presolved model:
• 19,111 constraints
• 1,000,825 variables (17 binaries)
• 4,299,915 non-zeros

Solving statistics:
• hit time limit after 72 search nodes
• final MIP gap is 2.40%
• used 15,543 simplex iterations per node

Biggest MIP that Gurobi can solve in 32 GB of RAM

Original model:
• 5,088,000 constraints
• 49,703,956 variables (55,139 binaries, 49,647,900 general integers)
• 328,860,900 non-zeros

Presolved model:
• 0 constraints
• 0 variables
• 0 non-zeros

Solving statistics:
• solved by heuristic (pure feasibility problem)
• 87.5 seconds
Mixed Integer Programming
Models with up to 10 million integer variables

Solving time vs. Number of integer variables

- Regular termination
- Out-of-memory
Mixed Integer Programming
Models with up to 1 million integer variables
Mixed Integer Programming

Models with up to 100,000 integer variables
Mixed Integer Programming
Models with up to 10,000 integer variables

No obvious relation between size (# integer variables) and solve time

What about the theoretical exponential worst-case runtime?
MIP is $\mathcal{NP}$-complete: Theory vs Practice

Models with up to 100 integer variables

Let’s zoom out a little bit again...

Worst-case bound for pure binary programs with evaluating 1 billion solutions per second: $2^n / 10^9$
MIP is \( \mathcal{NP} \)-complete: Theory vs Practice

Models with up to 10,000 integer variables

Worst-case bound for pure binary programs with evaluating 1 billion solutions per second: \( 2^n / 10^9 \)
MIP is \( NP \)-complete: Theory vs Practice
Models with up to 50 million integer variables

Worst-case bound for pure binary programs with evaluating 1 billion solutions per second: \( 2^n / 10^9 \)
MIP solvers employ various combinatorial and number theoretic sub-algorithms

Some of these algorithms have polynomial runtime

- Does this mean those will always be fast enough?
  - No! Even a quadratic algorithm is too slow in many situations!
  - For example, pair-wise comparison to identify parallel rows in a matrix $A \in \mathbb{R}^{m \times n}$ needs $O(m^2 n)$ operations

- Always think about big models!
- 1 million rows means about 500 billion pairs of rows to check
- Need an algorithm that is faster in practice, not necessarily in asymptotic behavior
- Need to include safeguards against quadratic overhead for corner cases
Consequences for MIP Solvers

- MIP solvers employ various combinatorial and number theoretic sub-algorithms.
- Some of these algorithms have **exponential runtime**
  - Does this mean those will never be useful?
    - No! Exponential worst-case runtime does not say anything about practical problem instances!
    - Often, we only need to solve small combinatorial problem instances to optimality.
    - In most cases, a heuristic that often finds good solutions is good enough.

- The algorithm design should be targeted towards practical problem instances
  - But always think about worst-case behavior to include safeguards in your code!
  - Quadratic loops are not always easy to spot in your code.
  - They constitute one of the most frequent “performance bugs” that we need to fix.
Many algorithms in our code do something with some variable, and then need to update some data for the variable’s neighbors.

Definition: in $A \in \mathbb{R}^{m \times n}$ two columns $j_1, j_2$ are neighbors if $A \cdot j_1 A \cdot j_2 \neq 0$

- Thus, the variables are neighbors if they appear together in at least one constraint.

Algorithm to find neighbors of $j_1$:
1. Set $N := \emptyset$
2. For each non-zero element $a_{i,j_1} \neq 0$ in $A \cdot j_1$:
   (a) For each non-zero element $a_{i,j_2} \neq 0$ in $A \cdot i$,
   (i) Set $N := N \cup \{j_2\}$

Now consider a constraint with $k$ non-zero elements
- If our algorithm touches each of the $k$ variables in the constraint and each time needs to find the neighbors of the current variable, we perform $k^2$ operations.
- No problem for $k = 1000$, but very bad for $k = 1,000,000$
Sparsity Patterns

30n20b8

bab2

lotsze

nw04

qap10

unitcal_7

pictures from miplib.zib.de
Sparsity Statistics

Full MIP test set

Median:
- 5.60 non-zeros/row
- 5.36 non-zeros/column

Average:
- 758.48 non-zeros/row
- 215.09 non-zeros/column
Sparsity Statistics
MIP test set without 5% of largest nz/row and 5% of largest nz/col ratios

Median:
• 5.13 non-zeros/row
• 5.09 non-zeros/column

Average:
• 18.65 non-zeros/row
• 9.01 non-zeros/column

Conclusion:
• Design your algorithms to be very fast with 2-30 non-zeros per row and 2-10 non-zeros per column
• But avoid runtime explosion for row or column lengths of 1000 and larger
Implementation Considerations

- Algorithms need to be implemented in C
  - Gurobi needs to support ancient and strange platforms like AIX, Solaris, or Windows 32
  - C compiles on every platform
    - Anything else (including C++) can get messy
- Algorithms often need to work on Gurobi’s internal data structures
  - If an algorithm is called frequently, we cannot afford translating our data structures into those that the algorithm works on
- Algorithms need to be tuned to the structures and sizes that appear in practical MIP models
- Gurobi provides malloc callbacks that Gurobi should use for its memory management
- Conclusion: need to implement all algorithms ourselves
  - Nice consequence: a lot of fun!
Combinatorial Algorithms

Median algorithm
Depth first search
Shortest path
Min cut / max flow
Minimum vertex separator
Max clique
Dynamic programming
Graph automorphism
Union find
Consider a single constraint linear program with bounds on the variables:

\[
\begin{align*}
\text{max} & \quad c^T x \\
\text{s.t.} & \quad a^T x \leq b \\
& \quad x_j \in [0, u_j] \quad \text{for all } j
\end{align*}
\]

This can be solved by sorting the elements: \( \frac{c_1}{a_1} \geq \frac{c_2}{a_2} \geq \cdots \geq \frac{c_n}{a_n} \)

Then, the solution is

\[
\begin{align*}
x_1 &= u_1, \ldots, x_{k-1} = u_{k-1}, x_k = \frac{1}{a_k} \left( b - \sum_{j=1}^{k-1} a_j u_j \right), x_{k+1} = \cdots = x_n = 0
\end{align*}
\]

But sorting is too slow: \( \mathcal{O}(n \cdot \log(n)) \)

Median algorithm can find critical element \( x_k \) in \( \mathcal{O}(n) \) steps
Median Algorithm

Dual simplex ratio test with bound flipping

- Dual pricing selects infeasible basic variable \( x_i \) to leave the basis
- Ratio test then selects non-basic variable \( x_j \) to enter the basis
  - Geometrically: follow ray in dual space until first dual constraint is hit
  - Finding first dual constraint that is hit means to find smallest value in list of “ratios”
- But instead of letting \( x_j \) enter the basis we may flip \( x_j \) to its opposite bound
  - Only possible if this flip in the primal space keeps \( x_i \) infeasible
  - If flip is valid, we can continue following this ray until next dual constraint is hit
- Thus, we have:
  - The infeasibility of \( x_i \) is our budget
  - For each ratio test candidate \( x_j \) we calculate how much of budget a bound flip costs
  - Simple algorithm would be to sort by ratio, then flip candidates until budget is exhausted and let the critical element enter the basis
  - Replace sorting by median algorithm to get linear runtime
- Performance impact on dual simplex algorithm:
  - 8.0% slower overall with sorting instead of median
  - 16.6% slower on models that take at least 10 seconds to solve
• Basic domain propagation for single constraint

\[ a_0x_0 + a^T x \leq b \]
\[ x_j \in [l_j, u_j] \text{ for all } j \]

• Relax constraint for other variables

\[ a_0x_0 + \min\{a^T x | x \in [l, u]\} \leq b \]

• Yields bound for \( x_0 \)
  • If \( a_0 > 0 \): \( x_0 \leq b' \)
  • If \( a_0 < 0 \): \( x_0 \geq b' \)
  • With \( b' = \frac{1}{a_0} (b - \min\{a^T x | x \in [l, u]\}) = \frac{1}{a_0} \left( b - \sum_{a_j > 0} a_j l_j - \sum_{a_j < 0} a_j u_j \right) \)

• Can we get stronger propagation by considering more than one constraint?
Domain propagation using two constraints

- Pick two constraints of the MIP

\[
\begin{align*}
    a_0 x_0 + a^T x & \leq b \\
    \bar{a}^T x & \leq \bar{b} \\
    x_j & \in [l_j, u_j] \quad \text{for all } j
\end{align*}
\]

that have some overlap (i.e., \(a^T \bar{a} \neq 0\))

- Relax constraint for other variables

\[
a_0 x_0 + \min \{ a^T x | \bar{a}^T x \leq \bar{b}, x \in [l, u] \} \leq b
\]

- Yields bound for \(x_0\)
  - If \(a_0 > 0\): \(x_0 \leq b'\)
  - If \(a_0 < 0\): \(x_0 \geq b'\)
    - With \(b' = \frac{1}{a_0} (b - \min \{ a^T x | \bar{a}^T x \leq \bar{b}, x \in [l, u] \})\)

- Inner problem \(\min \{ a^T x | \bar{a}^T x \leq \bar{b}, x \in [l, u] \}\) is a single constraint LP with bounds
Median Algorithm
Writing search tree nodes to disk

• If search tree grows too large, store uninteresting nodes to disk
  • Uninteresting: nodes with large dual bound
• Pick number of nodes we want to store to disk
• Nodes are not fully sorted, but stored in a heap
• Use median algorithm to find dual bound threshold in node heap
• Move all nodes with larger dual bound to disk, keep others in heap
Consider a MIP with disconnected components

\[
\begin{align*}
\min & \quad c^T x + \bar{c}^T \bar{x} \\
\text{s.t.} & \quad Ax \leq b \\
& \quad \bar{A} \bar{x} \leq \bar{b} \\
& \quad x, \bar{x} \in \mathbb{R}^n \\
& \quad x_j, \bar{x}_j \in \mathbb{Z} \quad \text{for all } j \in I, \bar{j} \in \bar{I}
\end{align*}
\]

• Solving this as a single MIP with branch-and-cut has worst-case runtime \(O(2^n + \bar{n})\)

• Solving the two MIPs separately has worst-case runtime \(O(2^n + 2\bar{n})\)

• Significant speed-up also occurs in practice
• How to find disconnected components in matrix $A$?
• Consider bipartite graph

Depth first search in this graph finds disconnected components of $A$
• Data structure: store $A$ twice
  • In row-wise sparse compressed form
  • In column-wise sparse compressed form
• Assume the bipartite matrix graph has an articulation point

• If this articulation point is a binary variable $y \in \{0,1\}$:
  • Solve smaller component as MIP for $y = 0$ and $y = 1$: optimal solutions $\bar{x}^0$ and $\bar{x}^1$
  • Aggregate variables: $\bar{x}_j := \bar{x}^0 + (\bar{x}^1 - \bar{x}^0)y$

• Find articulation points: Tarjan’s Algorithm for strongly connected components
  • Need to use non-recursive version of Tarjan (recursion depth may exceed stack size)
• With linear and SOS1 constraints you can model so-called indicator constraints

\[ z = 0 \rightarrow x_i = x_j \quad \text{or} \quad z = 0 \rightarrow x_i \neq x_j \]

for binary variables \( z, x_i \) and \( x_j \)

• Such constraints appear in some practical applications

• For example, MIPLIB model ‘toll-like’ is about the balanced subgraph problem
  • Appears in bioinformatics: finding monotone subsystems in gene regulatory networks
  • See http://miplib.zib.de/instance_details_toll-like.html and references
Consider a set of indicator constraints

\[ z_k = 0 \rightarrow x_{i_k} = x_{j_k} \text{ for } k \in E \]
\[ z_k = 0 \rightarrow x_{i_k} \neq x_{j_k} \text{ for } k \in U \]

Then, for an inequality indicator

\[ z_{s,t} = 0 \rightarrow x_s \neq x_t \]

and a path of constraints

\[ z_{s,k_1} = 0 \rightarrow x_s \equiv x_{k_1} \]
\[ z_{k_1,k_2} = 0 \rightarrow x_{k_1} \equiv x_{k_2} \]
\[ \vdots \]
\[ z_{k_n,t} = 0 \rightarrow x_{k_n} \equiv x_t \]

with an even number of inequality indicators, we can see that

\[ z_{s,t} + z_{s,k_1} + z_{k_1,k_2} + \cdots + z_{k_n,t} \geq 1 \]

is valid.
Cut separation algorithm for $z_{s,t} + z_{s,k_1} + z_{k_1,k_2} + \cdots + z_{k_n,t} \geq 1$

- Start with $z_{s,t}$ with fractional LP solution $z_{s,t}^* \notin \{0,1\}$
- Search for shortest path $s \rightarrow k_1 \rightarrow \cdots \rightarrow k_n \rightarrow t$
  - Lengths given by the LP values $z_{i,j}^*$
  - Only consider paths with even number of inequality indicators

Trick for even number of inequality indicators

- Two copies of graph: $G_1$ and $G_2$
- Equality indicators connect vertices within each copy
- Inequality indicators connect vertices between copies
- Nodes $s$ and $t$ only exist in $G_1$

Use Dijkstra’s algorithm to find shortest path
Shortest Path

Invalid cycle cuts

• Cut separation algorithm for $z_{s,t} + z_{s,k_1} + z_{k_1,k_2} + \cdots + z_{k_n,t} \geq 1$
  • Start with $z_{s,t}$ with fractional LP solution $z_{s,t}^* \not\in \{0,1\}$
  • Search for shortest path $s \rightarrow k_1 \rightarrow \cdots \rightarrow k_n \rightarrow t$
    • Lengths given by the LP values $z_{i,j}^*$
    • Only consider paths with even number of inequality indicators

• Trick for even number of inequality indicators
  • Two copies of graph: $G_1$ and $G_2$
  • Equality indicators connect vertices within each copy
  • Inequality indicators connect vertices between copies
  • Nodes $s$ and $t$ only exist in $G_1$

• Use Dijkstra’s algorithm to find shortest path
Shortest Path
Mod-2 and mod-k cuts

- Very similar construction possible to separate mod-2 and mod-k cuts
  - Caprara and Fischetti (1996): $\{0,\frac{1}{2}\}$-Chvátal-Gomory cuts
  - Caprara, Fischetti and Letchford (2000): On the separation of maximally violated mod-k cuts
  - Andreello, Caprara and Fischetti (2007): Embedding $\{0,\frac{1}{2}\}$-Cuts in a Branch-and-Cut Framework: A Computational Study
- But Gurobi uses different approach for these cuts
  - Gaussian LU factorization in mod-k space
  - Koster, Zymolka and Kutschka (2009): Algorithms to Separate $\{0,\frac{1}{2}\}$-Chvátal-Gomory Cuts
Shortest Path
Other applications in MIP

• Network heuristic
  • Find negative cost cycles to improve solution for problems with network structure

• Network simplex algorithm
  • Find negative cost cycles to detect negative reduced costs for pricing selection
Min-Cut / Max-Flow
Network cut separation

- Fixed charge network flow problem

- Flow conservation constraints: \( \sum_{a \in \delta^+(v)} f_a - \sum_{a \in \delta^-(v)} f_a = d_v \)
- Arc capacity constraints: \( f_a - c_a z_a \leq 0 \)
- Flow variables:
  \( f_a \geq 0 \)
- Arc selection variables:
  \( z_a \in \{0,1\} \)
**Min-Cut / Max-Flow**

Network cut separation

- Network cut:

- Capacity on network cut must be large enough to transport demand from $S$ to $T$ plus the flow that goes from $T$ back to $S$:

$$\sum_{a \in \delta^+(S)} c_a z_a - \sum_{a \in \delta^-(S)} f_a \geq \sum_{v \in S} d_v$$

- Dividing by any of the $c_a$ and applying mixed integer rounding yields cut-set inequalities
Min-Cut / Max-Flow
Network cut separation

• Heuristic to separate network cuts
  • Assign arc weights to be $w_a := s_a^* - |\pi_a^*|$
    • LP slack value $s_a^*$ for capacity constraint on arc $a$
    • Dual solution value $\pi_a^*$ for capacity constraint on arc $a$
  • Search for minimum weighted cut in resulting graph
  • Note that weights can be negative!
    • Minimum cut problem with negative weights is $NP$-hard

• Use heuristic for minimum cut problem
  • Try all single node sets $S = \{v\}$
  • Additionally, contract nodes in non-increasing order of weights $w_a$ until only 5 super nodes are left; then enumerate all cuts
    • Bienstock, Chopra, Günlük, Tsai (1998): Minimum cost capacity installation for multicommodity network flows
    • Günlük (1999): A branch and cut algorithm for capacitated network design problems
    • Achterberg and Raack (2010): The MCF-Separator – Detecting and Exploiting Multi-Commodity Flow Structures in MIPs
Minimum Vertex Separator

Nested-dissection fill-reducing ordering for interior point LP solver

• Runtime for interior point LP solver is dominated by cost of computing a sparse Cholesky factorization on $AA^T$

• Cost depends heavily on elimination order (ordering of rows of $A$)
  • Some orderings can lead to catastrophic fill-in

• Problem of finding optimal fill-reducing ordering is $\mathcal{NP}$-complete
  • Yannakakis (1981): Computing the Minimum Fill-In is NP-Complete
Minimum Vertex Separator

Nested-dissection fill-reducing ordering for interior point LP solver

- Adjacency graph in sparse Cholesky factorization
  - Simple correspondence between symmetric sparse matrix (structure) and adjacency graph
Minimum Vertex Separator
Nested-dissection fill-reducing ordering for interior point LP solver

- Adjacency graph in sparse Cholesky factorization
  - Simple correspondence between symmetric sparse matrix (structure) and adjacency graph
- Gaussian elimination produces cliques
Minimum Vertex Separator
Nested-dissection fill-reducing ordering for interior point LP solver

- Nested dissection ordering heuristic
  - Divide and conquer
  - Vertex separators disconnect the problem

Adjacent graph

Sparse matrix
Very common constraints in MIP are set packing constraints
\[ x_1 + \cdots + x_k \leq 1 \]
for binary variables \( x_j \).

Multiple set packing constraints can be merged, for example:
\[
\begin{align*}
  x_1 + x_2 & \leq 1 \\
  x_1 + x_3 & \leq 1 \\
  x_2 + x_3 & \leq 1
\end{align*}
\]

can be equivalently represented by
\[ x_1 + x_2 + x_3 \leq 1 \]

The latter has a much stronger LP relaxation than the former.
  • For example, \( x_1 = x_2 = x_3 = 0.5 \) is feasible for the former, but not for the latter.
• Consider stable set relaxation of a MIP
  • Graph $G = (V, E)$ with nodes $V$ being the (complemented) binary variables of the problem and edges $E = \{(i, j) | x_i, x_j \text{ share a set packing constraint}\}$

• For each set packing constraint $S \subseteq V$ find large clique $C \supseteq S$
  • Ideally, find maximum clique
  • Max clique is $\mathcal{NP}$-complete
  • Use heuristic to find large clique

• Replace $\sum_{j \in S} x_j \leq 1$ by $\sum_{j \in C} x_j \leq 1$

• Discard all set packing constraints with $S' \subseteq C$
Max Clique
Clique merging in presolve

• Many heuristics for max clique available
  • E.g., Robson (2001): Finding a maximum independent set in time $O(2^{n/4})$
• But: problem is not given as $G = (V, E)$
  • Instead, problem is given as $G = (V, C)$ with $C$ being a set of cliques
    • Edges $E$ implicitly given as all edges defined by cliques $C$
  • Consider set partitioning instances like nw04
    • Constraints with 50,000 variables imply >1 billion edges!
    • Cannot afford to create $G = (V, E)$ explicitly

picture from miplib.zib.de
Max Clique
Clique merging in presolve

• Gurobi heuristic is a greedy clique growing heuristic to obtain a maximum clique
  • Start by adding all variables of initial clique $S$ to $C$
    • Main operation: filter out nodes that are not neighbors of the recently added node $v$
      • Mark all cliques in which $v$ appears
      • Check for remaining candidates if they appear in one of the marked cliques
        • If not, remove candidate from list
    • Speed-up for main operation:
      • Consider nodes of starting clique in batches of size 32
      • Use bit logic for clique membership check
  • Then, add one or more of the remaining candidates to $C$
    • Add largest set of candidates that appear in a common clique
    • Safeguard: only process first 10 candidates to count clique cover number
      • Otherwise, too expensive for model with 4 million set packing constraints but only 6800 variables
Max Clique
Clique merging in presolve

• Main operation of Gurobi heuristic traverses columns of matrix
  • Find neighbors by processing the rows of the matrix
  • On average, this touches $\bar{l}_c + \bar{l}_c \cdot \bar{l}_r$ non-zero matrix entries
    • $\bar{l}_c$ and $\bar{l}_r$ being the average number of non-zeros in columns and rows
  • If all set packing constraints are of size 2, this means to touch $3\bar{l}_c$ non-zeros

• Separate clique merging algorithm specialized for short cliques
  • Considers only set packing constraints of size up to 100
  • Explicitly forms $G = (V, E)$, only storing one direction for each edge
  • Reduces memory access for size 2 cliques from $3\bar{l}_c$ to $\bar{l}_c$
    • Typically, translates into a runtime improvement of almost 3x

• See Achterberg, Bixby, Gu, Rothberg and Weninger (2019): Presolve Reductions in Mixed Integer Programming
Max Clique
Clique cuts

• Clique cut separation very similar to clique merging
• Differences:
  • Start with subset of clique
    • Only variables with $x_j^* > 0$
  • Weighted max clique
    • Maximize sum of LP solution values
    • Initially, only consider variables with $x_j^* > 0$
    • Final step is to grow clique further using variables with $x_j^* = 0$
Dynamic Programming
Knapsack coefficient strengthening

• Given a knapsack constraint

\[ a_0 x_0 + a_1 x_1 + \cdots + a_n x_n \leq b \]

with \( a_j > 0 \) and binary variables \( x_j \)

• Use dynamic programming to calculate

\[ \alpha^0 := \max \left\{ \{ \sum_{j=1}^{n} a_j x_j \mid x \in \{0,1\}^n \} \cap [0, b] \right\} \]

\[ \alpha^1 := \max \left\{ \{ \sum_{j=1}^{n} a_j x_j \mid x \in \{0,1\}^n \} \cap [0, b - a_0] \right\} \]

for the activity of the other variables \( j = 1, \ldots, n \), given \( x_0 = 0 \) or \( x_0 = 1 \)

• Lifting:
  - If \( d^1 := b - a_0 - \alpha^1 > 0 \): set \( a_0 := a_0 + d^1 \)
  - If \( d^0 := b - \alpha^0 > 0 \): set \( b := b - d^0 \) and \( a_0 := \max\{a_0 - d^0, 0\} \)
Example:

\[3x_0 + 4x_1 + 7x_2 + 8x_3 \leq 20\]

Use dynamic programming to calculate

\[\alpha^0 := \max\{4x_1 + 7x_2 + 8x_3 | x \in \{0,1\}^n \cap [0,20]\} = 19\]
\[\alpha^1 := \max\{\{4x_1 + 7x_2 + 8x_3 | x \in \{0,1\}^n \cap [0,17]\}\} = 15\]

Lifting:

- If \(d^1 := b - a_0 - \alpha^1 = 2 > 0\): set \(a_0 := a_0 + d^1 = 5\)
- If \(d^0 := b - \alpha^0 = 1 > 0\): set \(b := b - d^0 = 19\) and \(a_0 := \max\{a_0 - d^0, 0\} = 4\)

Result:

\[4x_0 + 4x_1 + 7x_2 + 8x_3 \leq 19\]
• Apply coefficient strengthening
  • on all knapsack constraints in an inner presolve loop
  • on all cutting planes generated during the search

• Thus, this is a very heavily used algorithm!

• Dynamic programming to calculate lifting values is $O(2^n)$
  • Apply only for knapsacks of length up to 10
  • Otherwise, use more complicated algorithm that
    • deals with a number of special cases,
    • calculates at most 64 different values inside the dynamic program, and
    • aborts if the required number of values exceeds 64
Dynamic Programming
Knapsack cover cut separation

• Given a knapsack constraint

\[ a_1 x_1 + \cdots + a_n x_n \leq b \]

with \( a_j > 0 \) and binary variables \( x_j \)

• A subset \( C \subseteq N := \{1, \ldots, n\} \) is called a cover if \( \sum_{j \in C} a_j > b \)

• Resulting cover cut: \( \sum_{j \in C} x_j \leq |C| - 1 \)

• Separation:
  • Set \( C^0 := \{ j | x_j^* = 0 \} \), \( C^1 := \{ j | x_j^* = 1 \} \), \( C^f := N \setminus C^0 \setminus C^1 \)
  • Find greedy minimum cover \( C \) for \( \sum_{j \in C^f} a_j x_j \leq b - \sum_{j \in C^1} a_j \)
  • Safeguard: only proceed if \( |C| \cdot n \leq 10^9 \)
  • Up-lift variables in \( C^f \setminus C \) to make cut stronger
  • Down-lift variables in \( C^1 \) to make cut valid for \( N \)
  • Up-lift variables in \( C^0 \) to make cut stronger
A bijection $f: \mathbb{R}^n \rightarrow \mathbb{R}^n$ is called a symmetry for a given MIP if
- it maps the feasible solution space $X$ of the MIP to itself: $f(X) = X$, and
- it preserves objective values: $c^T f(x) = c^T x$ for all $x \in X$

This definition based on feasible solution space $X$ is not practical, as deciding whether $X = \emptyset$ is \textit{NP}-complete.

In practice: consider permutations that leave constraints and objective invariant
- A permutation $\pi: N \rightarrow N$ of column indices is a \textit{formulation symmetry} if there exists a permutation $\sigma: M \rightarrow M$ of row indices such that
  - $\pi(l) = l$ (i.e., $\pi$ preserves integer variables),
  - $\pi(c) = c$,
  - $\sigma(b) = b$, and
  - $A_{\sigma(i), \pi(j)} = A_{i, j}$ for all $i \in M, j \in N$
Detecting formulation symmetries for MIP can be reduced to detecting graph automorphisms.

- Bipartite graph with nodes for constraints and variables, edges for non-zero coefficients.
- Constraint nodes are colored with right hand side values \( b_i \).
- Variable nodes are colored with objective values \( c_j \) (and integrality property).
- Edges are colored with matrix coefficients.
- Graph automorphism that respects colors is formulation symmetry of MIP.
• Complexity status of graph automorphism problem is still unknown
  • No polynomial algorithm known
  • Not proven to be \( \mathcal{NP} \)-hard
  • See Read and Corneil (1977): The graph isomorphism disease

• Efficient algorithms in practice exist
  • nauty
  • saucy
  • bliss

• Gurobi implements a variant of these algorithms

• See also Pfetsch and Rehn (2019): A computational comparison of symmetry handling methods for mixed integer programs
Graph Automorphism
Symmetry detection in Gurobi

- Maintain two sets of partitions for constraints and variables
  - \( \Sigma \) and \( \Pi \) to group constraints and variables that could potentially be in same orbit
  - \( \overline{\Sigma} \) and \( \overline{\Pi} \) to group constraints and variables that are definitely in the same orbit
- Initially, \( \overline{\Sigma} \) and \( \overline{\Pi} \) are defined by node colors, \( \Sigma \) and \( \Pi \) are all singletons
- Recursively refine \( \overline{\Sigma} \) and \( \overline{\Pi} \) using hash values
  - calculated from hash values of neighbor nodes
- If fix point is reached, branch on a non-singleton part of \( \overline{\Sigma} \) or \( \overline{\Pi} \)
  - Failed branch refines partitions and thus hash values
  - Leaf branching node corresponds to valid symmetry generator and updates \( \Sigma \) and \( \Pi \)
- Perform branching with backtracking until
  - \( \overline{\Sigma} = \Sigma \) and \( \overline{\Pi} = \Pi \) (generators to produce all symmetries have been found), or
  - a work limit has been hit (generators produce a subset of the symmetries)
• Important tricks to get good performance in practice
  • Sparse updates of data structures
    • Only touch those constraints and variables in refinement that have changed
    • When splitting a partition class, assign new label to smaller part
  • Special treatment of singleton partition classes
    • Remove them from graph after hash update, as their hashes won't change anymore
  • Use very good hash function to avoid hash collisions
  • Initially, check whether old symmetry generators are still valid
    • If we search for symmetry again after some problem changes
  • Check work limits regularly to avoid bad corner cases

• Why care?
  • Exploiting symmetry yields ~20% performance improvement overall
  • ~2x speed-up on affected models
  • See Achterberg and Wunderling (2013): Mixed Integer Programming: Analyzing 12 Years of Progress
Consider a symmetry generator \( g: N \to N \) that is

- non-overlapping
  - No \( x_j \) appears in the same constraint as \( x_{g(j)} \)
  - or that does not affect integer variables
    - For all \( j \in I \) we have \( g(j) = j \)

Then we can aggregate all variables according to the generator:

\[
x_j := x_{g(j)}
\]

- Each symmetry generator extends sets of equivalent variables
- This can be efficiently recorded with a union find data structure
Other Interesting Algorithms

- Sorting
- Euclidean algorithm
- Hashing
- Random number generation

... not covered today