DIP and DipPy: Towards a Decomposition-based MILP Solver

Ted Ralphs
Lehigh University
Matthew Galati
SAS Institute
Jiadong Wang
Lehigh University

International Symposium on Mathematical Programming, August 2012

Thanks: Work supported in part by the National Science Foundation
Outline

1. Methods
   - Traditional
   - Generic

2. Software

3. Generic Interfaces
   - DipPy
   - C++
   - Command Line

4. Computational Experiments
   - Exploiting Structure
   - Detecting Structure
   - Parallelizing

5. Current and Future Research
The Basic Setting

Integer Linear Program: Minimize/Maximize a linear \textit{objective function} over a (discrete) set of \textit{solutions} satisfying specified \textit{linear constraints}.

\[
z_{\text{IP}} = \min_{x \in \mathbb{Z}^n} \left\{ c^\top x \mid A'x \geq b', A''x \geq b'' \right\}
\]

\[
z_{\text{LP}} = \min_{x \in \mathbb{R}^n} \left\{ c^\top x \mid A'x \geq b', A''x \geq b'' \right\}
\]

\[
\mathcal{P} = \text{conv}\{x \in \mathbb{Z}^n \mid A'x \geq b', A''x \geq b''\}
\]
What is the Goal of Decomposition?

**Basic Idea:** Exploit knowledge of the underlying structural components of model to improve the bound.

Many complex models are built up from multiple underlying substructures.
- Subsystems linked by global constraints.
- Complex combinatorial structures obtained by combining simple ones.

We want to exploit knowledge of efficient methodology for substructures.
This can be done in two primary ways (with many variants).
- Identify independent subsystems.
- Identify subsets of constraints that can be dealt with efficiently.
Outline

1. Methods
   - Traditional
   - Generic

2. Software

3. Generic Interfaces
   - DipPy
   - C++
   - Command Line

4. Computational Experiments
   - Exploiting Structure
   - Detecting Structure
   - Parallelizing

5. Current and Future Research
The Decomposition Principle in Integer Programming

**Basic Idea:** By leveraging our ability to solve the optimization/separation problem for a relaxation, we can improve the bound yielded by the LP relaxation.

\[
\begin{align*}
\text{z}_{\text{IP}} &= \min_{x \in \mathbb{Z}^n} \left\{ c^\top x \mid A' x \geq b', A'' x \geq b'' \right\} \\
\text{z}_{\text{LP}} &= \min_{x \in \mathbb{R}^n} \left\{ c^\top x \mid A' x \geq b', A'' x \geq b'' \right\} \\
\text{z}_{\text{D}} &= \min_{x \in \mathcal{P}'} \left\{ c^\top x \mid A'' x \geq b'' \right\} \\
\text{z}_{\text{IP}} &\geq \text{z}_{\text{D}} \geq \text{z}_{\text{LP}}
\end{align*}
\]

**Assumptions:**
- OPT(\(\mathcal{P}, c\)) and SEP(\(\mathcal{P}, x\)) are “hard”
- OPT(\(\mathcal{P}', c\)) and SEP(\(\mathcal{P}', x\)) are “easy”
- \(Q''\) can be represented explicitly (description has polynomial size)
- \(\mathcal{P}'\) must be represented implicitly (description has exponential size)
Overview of Methods

Cutting Plane Method (CPM)

**CPM** combines an *outer* approximation of $\mathcal{P}'$ with an explicit description of $\mathcal{Q}''$

- **Master:** $z_{\text{CP}} = \min_{x \in \mathbb{R}^n} \{ c^\top x \mid Dx \geq d, A''x \geq b'' \}$
- **Subproblem:** $\text{SEP}(\mathcal{P}', x_{\text{CP}})$

Dantzig-Wolfe Method (DW)

**DW** combines an *inner* approximation of $\mathcal{P}'$ with an explicit description of $\mathcal{Q}''$

- **Master:** $z_{\text{DW}} = \min_{\lambda \in \mathbb{R}_{+}^E} \{ c^\top (\sum_{s \in \mathcal{E}} s\lambda_s) \mid A''(\sum_{s \in \mathcal{E}} s\lambda_s) \geq b'', \sum_{s \in \mathcal{E}} \lambda_s = 1 \}$
- **Subproblem:** $\text{OPT}(\mathcal{P}', c^\top - u_{\text{DW}}^\top A'')$

Lagrangian Method (LD)

**LD** iteratively produces single extreme points of $\mathcal{P}'$ and uses their violation of constraints of $\mathcal{Q}''$ to converge to the same optimal face of $\mathcal{P}'$ as CPM and DW.

- **Master:** $z_{\text{LD}} = \max_{u \in \mathbb{R}_{+}^m} \{ \min_{s \in \mathcal{E}} \{ c^\top s + u^\top (b'' - A''s) \} \}$
- **Subproblem:** $\text{OPT}(\mathcal{P}', c^\top - u_{\text{LD}}^\top A'')$
**Common Threads**

The **LP bound** is obtained by optimizing over the intersection of two explicitly defined polyhedra.

\[ z_{\text{LP}} = \min_{x \in \mathbb{R}^n} \{ c^T x \mid x \in Q' \cap Q'' \} \]

The **decomposition bound** is obtained by optimizing over the intersection of one explicitly defined polyhedron and one implicitly defined polyhedron.

\[ z_{\text{CP}} = z_{\text{DW}} = z_{\text{LD}} = z_{\text{D}} = \min_{x \in \mathbb{R}^n} \{ c^T x \mid x \in P' \cap Q'' \} \geq z_{\text{LP}} \]

Traditional decomp-based bounding methods contain two primary steps

- **Master Problem**: Update the primal/dual solution information
- **Subproblem**: Update the approximation of \( P' \): \( \text{SEP}(P', x) \) or \( \text{OPT}(P', c) \)

**Integrated decomposition methods** further improve the bound by considering two implicitly defined polyhedra whose descriptions are iteratively refined.

- **Price-and-Cut** (PC)
- **Relax-and-Cut** (RC)
- **Decompose-and-Cut** (DC)
Traditionally, decomposition-based branch-and-bound methods have required extensive problem-specific customization.

- identifying the decomposition (which constraints to relax);
- formulating and solving the subproblem (either optimization or separation over $P'$);
- formulating and solving the master problem; and
- performing the branching operation.

However, it is possible to replace these components with generic alternatives.

- The decomposition can be identified externally by analyzing the matrix or through a modeling language.
- The subproblem can be solved with a generic MILP solver.
- The branching can be done in the original compact space.

The remainder of this talk focuses on our recent efforts to develop a completely generic decomposition-based MILP solver.
The key to the implementation of this unified framework is that we always maintain a representation of the problem in the compact space.

This allows us to employ most of the usual techniques used in LP-based branch and bound without modification, even in this more general setting.

There are some challenges related to this approach that we are still working on.

- Gomory cuts
- Preprocessing
- Identical subproblems
- Strong branching

Allowing the user to express all methods in the compact space is extremely powerful when it comes to modeling language support.

It is important to note that DIP currently assumes the existence of a formulation in the compact space.

We are working on relaxing this assumption, but this means the loss of the fully generic implementation of some techniques.
Many difficult MILPs have a block structure, but this structure is not part of the input (MPS) or is not exploitable by the solver.

In practice, it is common to have models composed of independent subsystems coupled by global constraints.

The result may be models that are highly symmetric and difficult to solve using traditional methods, but would be easy to solve if the structure were known.

$$\begin{pmatrix}
A''_1 & A''_2 & \cdots & A''_\kappa \\
A'_1 & \ & \ & \\
\ & A'_2 & \ & \\
\ & \ & \cdots & \\
\ & \ & \ & A'_\kappa
\end{pmatrix}$$
Automatic Structure Detection

- For unstructured problems, block structure may be detected automatically.
- This is done using hypergraph partitioning methods.
- We map each row of the original matrix to a hyperedge and the nonzero elements to nodes in a hypergraph.
- Hypergraph partitioning results in identification of the blocks in a singly-bordered block diagonal matrix.
Hidden Block Structure

MIPLIB2003 instance: p2756

nz = 8937
Hidden Block Structure

Instance p2756 with 10 blocks partitioning
Generic Branching

- By default, we branch on variables in the compact space.
- In PC, this is done by mapping back to the compact space $\hat{x} = \sum_{s \in \mathcal{E}} s \hat{\lambda}_s$.
- Variable branching in the compact space is constraint branching in the extended space.
- This idea makes it possible to define generic branching procedures.

Node 1:
\[ 4\lambda_{(4,1)} + 5\lambda_{(5,5)} + 2\lambda_{(2,1)} + 3\lambda_{(3,4)} \leq 2 \]

Node 2:
\[ 4\lambda_{(4,1)} + 5\lambda_{(5,5)} + 2\lambda_{(2,1)} + 3\lambda_{(3,4)} \geq 3 \]
Outline

1 Methods
   - Traditional
   - Generic

2 Software

3 Generic Interfaces
   - DipPy
   - C++
   - Command Line

4 Computational Experiments
   - Exploiting Structure
   - Detecting Structure
   - Parallelizing

5 Current and Future Research
The use of decomposition methods in practice is hindered by a number of serious drawbacks.

- **Implementation is difficult**, usually requiring development of sophisticated customized codes.
- Choosing an algorithmic strategy requires **in-depth knowledge** of theory and strategies are **difficult to compare empirically**.
- The powerful techniques modern solvers use to solve integer programs are **difficult to integrate** with decomposition-based approaches.

**DIP and CHiPPS** are two frameworks that together allow for easier implementation of decomposition approaches.

- **CHiPPS** (COIN High Performance Parallel Search Software) is a flexible library hierarchy for implementing parallel search algorithms.
- **DIP** (Decomposition for Integer Programs) is a framework for implementing decomposition-based bounding methods.
- **DIP with CHiPPS** is a full-blown branch-and-cut-and-price framework in which details of the implementation are hidden from the user.

DIP can be accessed through a modeling language or by providing a model with notated structure.
DIP Framework: Feature Overview

- One interface to all algorithms: CP, DW, LD, PC, RC—change method by switching parameters.
- **Automatic reformulation** allows users to specify methods in the compact (original) space.
- Built on top of the OSI interface, so easy to swap solvers (simplex to interior point).
- Novel options for cut generation
  - Can utilize CGL cuts in all algorithms (separate from original space).
  - Can utilize structured separation (efficient algorithms that apply only to vectors with special structure (integer) in various ways).
  - Can separate from $P'$ using subproblem solver (DC).
- Easy to combine different approaches
  - Column generation based on *multiple algorithms* or *nested subproblems* can be easily defined and employed.
  - Bounds based on *multiple model/algorithm* combinations.
- Active LP compression, variable and cut pool management.
The base class `DecompApp` provides an interface for user to define the application-specific components of their algorithm.

- **Define the model(s)**
  - `setModelObjective(double * c)`: define $c$
  - `setModelCore(DecompConstraintSet * model)`: define $Q''$
  - `setModelRelaxed(DecompConstraintSet * model, int block)`: define $Q'$ [optional]

- `solveRelaxed()`: define a method for $\text{OPT}(P', c)$ [optional, if $Q'$, CBC is built-in]

- `generateCuts()`: define a method for $\text{SEP}(P', x)$ [optional, CGL is built-in]

- `isUserFeasible()`: is $\hat{x} \in P$? [optional, if $P = \text{conv}(P' \cap Q'' \cap \mathbb{Z})$]

All methods have appropriate defaults but are **virtual** and may be overridden.

The base class `DecompAlgo` provides the shell (init / master / subproblem / update).

- Each of the methods described has derived default implementations `DecompAlgoX : public DecompAlgo` which are accessible by any application class, allowing full flexibility.

- New, hybrid or extended methods can be easily derived by overriding the various subroutines, which are called from the base class.
Outline

1 Methods
   - Traditional
   - Generic

2 Software

3 Generic Interfaces
   - DipPy
   - C++
   - Command Line

4 Computational Experiments
   - Exploiting Structure
   - Detecting Structure
   - Parallelizing

5 Current and Future Research
DipPy provides an interface to DIP through the modeling language PuLP.

PuLP is a modeling language that provides functionality similar to other modeling languages.

It is built on top of Python so you get the full power of that language for free.

PuLP and DipPy are being developed by Stuart Mitchell and Mike O’Sullivan in Auckland and are part of COIN.

Through DipPy, a user can

- Specify the model and the relaxation, including the block structure.
- Implement methods (coded in Python) for solving the relaxation, generating cuts, custom branching.

With DipPy, it is possible to code a customized column-generation method from scratch in a few hours.

This would have taken months with previously available tools.
Example: Generalized Assignment Problem

- The problem is to find a minimum cost assignment of \( n \) tasks to \( m \) machines such that each task is assigned to one machine subject to capacity restrictions.
- A binary variable \( x_{ij} \) indicates that machine \( i \) is assigned to task \( j \). \( M = 1, \ldots, m \) and \( N=1, \ldots, n \).
- The cost of assigning machine \( i \) to task \( j \) is \( c_{ij} \)

### Generalized Assignment Problem (GAP)

\[
\begin{align*}
\text{min} & \quad \sum_{i \in M} \sum_{j \in N} c_{ij} x_{ij} \\
\sum_{j \in N} w_{ij} x_{ij} & \leq b_i \quad \forall i \in M \\
\sum_{i \in M} x_{ij} & = 1 \quad \forall j \in N \\
x_{ij} & \in \{0,1\} \quad \forall i, j \in M \times N
\end{align*}
\]
Creating GAP model in DipPy

```python
prob = dippy.DipProblem("GAP", LpMinimize)

# objective
prob += lpSum(assignVars[m][t] * COSTS[m][t] for m, t in MACHINES_TASKS), "min"

# machine capacity (knapsacks, relaxation)
for m in MACHINES:
    prob.relaxation[m] +=
    lpSum(assignVars[m][t] * RESOURCE_USE[m][t] for t in TASKS) <= CAPACITIES[m]

# assignment
for t in TASKS:
    prob += lpSum(assignVars[m][t] for m in MACHINES) == 1

prob.relaxed_solver = relaxed_solver

dippy.Solve(prob)
```
def relaxed_solver(prob, machine, redCosts, convexDual):
    # get tasks which have negative reduced
    task_idx = [t for t in TASKS if redCosts[assignVars[machine][t]] < 0]
    vars = [assignVars[machine][t] for t in task_idx]
    obj = [-redCosts[assignVars[machine][t]] for t in task_idx]
    weights = [RESOURCE_USE[machine][t] for t in task_idx]

    z, solution = knapsack01(obj, weights, CAPACITIES[machine])
    z = -z

    # get sum of original costs of variables in solution
    orig_cost = sum(prob.objective.get(vars[idx]) for idx in solution)
    var_values = [(vars[idx], 1) for idx in solution]

    dv = dippy.DecompVar(var_values, z-convexDual, orig_cost)

    # return, list of DecompVar objects
    return [dv]
GAP in DipPy

DipPy Auxiliary Methods

```python
def solve_subproblem(prob, index, redCosts, convexDual):
    ...
    z, solution = knapsack01(obj, weights, CAPACITY)
    ...
    return []
prob.relaxed_solver = solve_subproblem

def knapsack01(obj, weights, capacity):
    ...
    return c[n-1][capacity], solution

def first_fit(prob):
    ...
    return bvs

def one_each(prob):
    ...
    return bvs
prob.init_vars = first_fit

def choose_antisymmetry_branch(prob, sol):
    ...
    return ([], down_branch_ub, up_branch_lb, [])
prob.branch_method = choose_antisymmetry_branch

def generate_weight_cuts(prob, sol):
    ...
    return new_cuts
prob.generate_cuts = generate_weight_cuts

def heuristics(prob, xhat, cost):
    ...
    return sols
prob.heuristics = heuristics
dippy.Solve(prob, {
    'doPriceCut': '1',
})
```
DIP createModels for GAP example

```cpp
void GAP_De compApp::createModels(){

    //get information about this problem instance
    int nTasks = m_instance.getNTasks(); //n
    int nMachines = m_instance.getNMachines(); //m
    const int * profit = m_instance.getProfit();
    int nCols = nTasks * nMachines;

    //construct the objective function
    m_objective = new double[nCols];
    for (i = 0; i < nCols; i++) { m_objective[i] = profit[i]; }
    setModelObjective(m_objective);

    DecompConstraintSet * modelCore = new DecompConstraintSet();
    createModelPartAP(modelCore);
    setModelCore(modelCore);

    for (i = 0; i < nMachines; i++){
        DecompConstraintSet * modelRelax = new DecompConstraintSet();
        createModelPartKP(modelRelax, i);
        setModelRelax(modelRelax, i);
    }
}
```
GAP in C++

DIP solveRelaxed for GAP example

```cpp
DecompSolverStatus GAP_DecompApp::solveRelaxed(const int whichBlock,
                                               const double * costCoeff,
                                               DecompVarList & newVars)
{
  DecompSolverStatus status = DecompSolStatNoSolution;
  if (!m_appParam.UsePisinger) { return status; }

  vector<int> solInd;
  vector<double> solEls;
  double varCost = 0.0;
  const double * costCoeffB = costCoeff + getOffsetI(whichBlock);

  status = m_knap[whichBlock]->solve(whichBlock, costCoeffB,
                                       solInd, solEls, varCost);
  DecompVar * var = new DecompVar(solInd, solEls, varCost, whichBlock);
  newVars.push_back(var);
  return status;
}
```

Ralphs, Galati, Wang
Decomposition Methods for Integer Linear Programming
```c
int main(int argc, char ** argv) {
    // create the utility class for parsing parameters
    UtilParameters utilParam(argc, argv);
    bool doCut = utilParam.GetSetting("doCut", true);
    bool doPriceCut = utilParam.GetSetting("doPriceCut", false);
    bool doRelaxCut = utilParam.GetSetting("doRelaxCut", false);

    // create the user application (a DecompApp)
    GAP_DecompApp gapApp(utilParam);

    // create the CPM/PC/RC algorithm objects (a DecompAlgo)
    DecompAlgo * algo = NULL;
    if (doCut) algo = new DecompAlgoC(&gapApp, &utilParam);
    if (doPriceCut) algo = new DecompAlgoPC(&gapApp, &utilParam);
    if (doRelaxCut) algo = new DecompAlgoRC(&gapApp, &utilParam);

    // create the driver AlpsDecomp model
    AlpsDecompModel gap(utilParam, algo);

    // solve
    gap.solve();
}
```
A third way of getting a model into the generic solver module is to read it from a file as usual.

The decomposition can be identified either by

- specifying the block structure explicitly by also reading a block file; or
- using the automatic block detection algorithm to identify the blocks.
Outline

1. Methods
   - Traditional
   - Generic

2. Software

3. Generic Interfaces
   - DipPy
   - C++
   - Command Line

4. Computational Experiments
   - Exploiting Structure
   - Detecting Structure
   - Parallelizing

5. Current and Future Research
Instances: ATM cash management, Wedding planner problem, MIPLIB

Computational environment: single nodes of a cluster, each of which has two 8 core processors, each running at 2 GHz and with 512 KB cache, 32 GB of shared memory,

Experiments

- DIP vs. CPLEX: Can decomposition be better than branch and cut?
- Automatic structure detection
  - Can we effectively use hypergraph partitioning to find structure?
  - How good is the bound?
  - How can we measure the goodness of a decomposition a priori?
  - How many blocks?
  - Which hypergraph partitioner is most effective?
- Can we exploit the block structure to parallelize the subproblem solve?

We use bound at the root node as our primary measure for comparison in most of the following experiments.

Caveat: these results are preliminary and their purpose is more to raise questions than answer them!
Wedding Planner Problem: CPLEX vs DIP

![Graph showing clock time (s) vs number of guests for CPLEX and DIP. The graph compares the performance of CPLEX 12.2 and DIP for different numbers of guests, highlighting the differences in computational time.]
We use heuristics to find a $K$-way hypergraph partition that minimizes the size of the resulting cut.

For midsize MIPLIB instances, it takes at most a few seconds.

Solvers

- hMETIS, Karypis, University of Minnesota
- PaTOH, Çatalyürek, Ohio State University

Comparisons: PaToH tends to be faster, but hMETIS produces better decompositions.
Quality Measures for Decomposition

- The goal of the partitioning is to have a "good decomposition."
- Generally, we judge goodness in terms of bound and computation time.
- There is a potential tradeoff involving the number of blocks, the number of linking rows, and the distribution of integer variables.
- We would like to be able to identify a good decomposition based on easily identified features.

### Potential Features

- The fraction of nonzero elements in the matrix appearing in the coupling rows ($\alpha$),
- The fraction of nonzeros appearing in the coupling rows that are in integer columns ($\beta$),
- The fraction of the nonzeros in integer columns in the matrix that appear in coupling rows ($\gamma$),
- The average fraction of the nonzeros in each block that are in integer columns ($\eta$),
- The standard deviation of the fraction of integer elements in the blocks ($\theta$).
Relationship between Features

![Graph showing the relationship between DWD optimality gap closed (%) and Percentage (%). The graph includes various data points represented by different symbols and markers, indicating the relationship between the two variables.]
A Measure for Decomposition Quality

\[ \Pi = (1 - \min(\alpha, \gamma)) \times 100\% , \]
We have now seen the features that are considered “important” in identifying a good decomposition.

How do we encourage the partitioner to give us such a decomposition?

With respect to the underlying graph, the partitioner has two goals.

- The weight of the cut should be minimized.
- The partition should be “balanced.”

The first goal essentially corresponds to minimizing the number of coupling rows.

The second goal corresponds to balancing the size of the blocks.

We can affect the behavior of the algorithm by assigning weights to the nodes and hyperedges.
Choices, Choices...

Instance a1c1s1 with 2 blocks partitioning

Instance a1c1s1 with 6 blocks partitioning

Instance a1c1s1 with 6 blocks partitioning

Instance a1c1s1 with 8 blocks partitioning
Effect of Number of Blocks

Figure: Relationship between number of blocks and optimality gap closed
Number of Blocks, DW Bound, and Computation Time

![Graph showing the relationship between number of blocks and DW bound for two MIPLIB instances, pg and machrophage. The graphs display the decrease in DW bound and computational time as the number of blocks increases.]
Weighting the Hyperedges and Nodes

- Unit weight as input of hypergraph partitioner already gives nice results, but can we do better?

- We have already seen that it is advantageous for the rows involving more integer variables to be in the blocks.

- We assign higher weight to nodes that correspond to integer variables and to hyperedges which correspond to rows containing integer elements.
Bound at the Root Node

![Graph showing the relationship between Optimality gap with LP bound and Optimality gap with DW bound. The graph includes a dashed line representing the correlation.](image-url)
Optimality Gap Closed

- Optimality gap closed by Dantzig–Wolfe decomposition
- Optimality gap closed by cutting–plane methods

Ralphs, Galati, Wang
Decomposition Methods for Integer Linear Programming 43/50
Parallelizing DIP

- Multi-core architecture are pervasive.
- DW decomposition of problems with block structure is naturally tailored to parallel computation.
- Here, we investigate the extent to which computation times can be improved by solving the subproblems in parallel.
- We measure time to process the root node.
- Our performance measure is:

$$E = \frac{T_1}{T_p \times p},$$

where $T_p$ is defined as the wallclock time using $p$ threads.
ATM Problem: Efficiency vs Cores

![Graph showing parallel efficiency vs number of threads for different number of cores. The graph demonstrates how parallel efficiency changes as the number of threads increases.]

Ralphs, Galati, Wang

Decomposition Methods for Integer Linear Programming
Wedding Planner Problem: Efficiency vs Cores

![Graph showing parallel efficiency vs number of threads]

- Parallel efficiency
- Number of threads

Ralphs, Galati, Wang
Decomposition Methods for Integer Linear Programming 46/50
Parallel Efficiency for Structured Problems

- We observe that utilizing multiple cores does reduce computational time.
  - As usual, there is a loss of parallel efficiency as we increase the number of threads.
    - Solution of the master problem is sequential
    - Idle time coming from differences in variability of solution time for different blocks.
- We use the normalized standard deviation (NS) of subproblem computational time at the first iteration to assess the variability of subproblem solution times.
  - For ATM and Wedding planner problems, the computational time is mainly allocated to solving subproblems.
  - The NS (average around 0.9) is higher for ATM problems than that (average around 0.2) for Wedding problem, which is why the parallel efficiency for Wedding problem is higher than that for ATM problems.
Using automatic hypergraph partitioning, we experimented with using DIP to solve generic (unstructured) MILPs in parallel.

<table>
<thead>
<tr>
<th>instance</th>
<th>LP%</th>
<th>DW%</th>
<th>ns</th>
<th>$T_1$</th>
<th>$P_1$</th>
<th>$E_4$</th>
<th>$P_4$</th>
<th>$E_8$</th>
<th>$P_8$</th>
<th>$E_{16}$</th>
<th>$P_{16}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>protfold</td>
<td>35</td>
<td>16</td>
<td>6.6</td>
<td>133</td>
<td>0.28</td>
<td>0.36</td>
<td>0.12</td>
<td>0.23</td>
<td>0.42</td>
<td>0.1</td>
<td>0.74</td>
</tr>
<tr>
<td>pp08aCUTS</td>
<td>25</td>
<td>7</td>
<td>0.59</td>
<td>1.28</td>
<td>0.91</td>
<td>0.52</td>
<td>0.84</td>
<td>0.28</td>
<td>0.79</td>
<td>0.07</td>
<td>0.77</td>
</tr>
<tr>
<td>pp08a</td>
<td>62</td>
<td>9</td>
<td>0.28</td>
<td>0.65</td>
<td>0.93</td>
<td>0.4</td>
<td>0.87</td>
<td>0.28</td>
<td>0.83</td>
<td>0.14</td>
<td>0.87</td>
</tr>
<tr>
<td>pg5_34</td>
<td>16</td>
<td>0</td>
<td>0.63</td>
<td>79</td>
<td>0.92</td>
<td>0.58</td>
<td>0.79</td>
<td>0.33</td>
<td>0.77</td>
<td>0.21</td>
<td>0.79</td>
</tr>
<tr>
<td>machrophage</td>
<td>100</td>
<td>2</td>
<td>4.0</td>
<td>488</td>
<td>0.99</td>
<td>0.62</td>
<td>0.99</td>
<td>0.38</td>
<td>0.98</td>
<td>0.24</td>
<td>0.98</td>
</tr>
<tr>
<td>manna81</td>
<td>1</td>
<td>0</td>
<td>0.37</td>
<td>38</td>
<td>0.79</td>
<td>0.4</td>
<td>0.67</td>
<td>0.17</td>
<td>0.49</td>
<td>0.15</td>
<td>0.49</td>
</tr>
<tr>
<td>mkc</td>
<td>9</td>
<td>4</td>
<td>1.3</td>
<td>62</td>
<td>0.9</td>
<td>0.52</td>
<td>0.84</td>
<td>0.19</td>
<td>0.84</td>
<td>0.16</td>
<td>0.8</td>
</tr>
<tr>
<td>modglob</td>
<td>1</td>
<td>1</td>
<td>0.49</td>
<td>4.96</td>
<td>0.94</td>
<td>0.59</td>
<td>0.78</td>
<td>0.74</td>
<td>0.72</td>
<td>0.17</td>
<td>0.73</td>
</tr>
<tr>
<td>10teams</td>
<td>1</td>
<td>1</td>
<td>3.36</td>
<td>103</td>
<td>0.6</td>
<td>0.15</td>
<td>0.45</td>
<td>0.1</td>
<td>0.54</td>
<td>0.03</td>
<td>0.31</td>
</tr>
<tr>
<td>cap6000</td>
<td>0</td>
<td>0</td>
<td>0.68</td>
<td>1.43</td>
<td>0.67</td>
<td>0.52</td>
<td>0.44</td>
<td>0.29</td>
<td>0.36</td>
<td>0.19</td>
<td>0.28</td>
</tr>
<tr>
<td>disctom</td>
<td>0</td>
<td>0</td>
<td>3.3</td>
<td>173</td>
<td>0.23</td>
<td>0.27</td>
<td>0.14</td>
<td>0.12</td>
<td>0.16</td>
<td>0.08</td>
<td>0.13</td>
</tr>
<tr>
<td>fixnet6</td>
<td>70</td>
<td>20</td>
<td>1.31</td>
<td>0.73</td>
<td>0.88</td>
<td>0.35</td>
<td>0.78</td>
<td>0.26</td>
<td>0.72</td>
<td>0.15</td>
<td>0.68</td>
</tr>
</tbody>
</table>

Variability of the subproblem solve times is larger than with the structured problems and we see an efficiency loss.
Outline

1 Methods
   - Traditional
   - Generic

2 Software

3 Generic Interfaces
   - DipPy
   - C++
   - Command Line

4 Computational Experiments
   - Exploiting Structure
   - Detecting Structure
   - Parallelizing

5 Current and Future Research
Current Research

- **Block structure** *(Important!)*
  - Identical subproblems for eliminating symmetry
  - Better automatic detection

- **Parallelism**
  - Parallel solution of subproblems with block structure
  - Parallelization of search using ALPS
  - Solution of multiple subproblems or generation of multiple solutions in parallel.
  - Generation of decomposition cuts for various relaxed polyhedra - diversity of cuts

- **Branch-and-Relax-and-Cut**: Computational focus thus far has been on CPM/DC/PC

- **General algorithmic improvements**
  - Improvements to warm-starting of node solves
  - Improved search strategy
  - Improved branching (strong branching, pseudo-cost branching, etc.)
  - Better dual stabilization
  - Improved generic column generation (multiple columns generated per round, etc)

- **Addition of generic MILP techniques**
  - Heuristics, branching strategies, presolve
  - Gomory cuts in Price-and-Cut
Current Research

- **Block structure** *(Important!)*
  - Identical subproblems for eliminating symmetry
  - Better automatic detection

- **Parallelism**
  - Parallel solution of subproblems with block structure
  - Parallelization of search using ALPS
  - Solution of multiple subproblems or generation of multiple solutions in parallel.
  - Generation of decomposition cuts for various relaxed polyhedra - diversity of cuts

- **Branch-and-Relax-and-Cut**: Computational focus thus far has been on CPM/DC/PC

- **General algorithmic improvements**
  - Improvements to warm-starting of node solves
  - Improved search strategy
  - Improved branching (strong branching, pseudo-cost branching, etc.)
  - Better dual stabilization
  - Improved generic column generation (multiple columns generated per round, etc)

- **Addition of generic MILP techniques**
  - Heuristics, branching strategies, presolve
  - Gomory cuts in Price-and-Cut
Current Research

**Block structure** *(Important!)*
- Identical subproblems for eliminating symmetry
- Better automatic detection

**Parallelism**
- Parallel solution of subproblems with block structure
- Parallelization of search using ALPS
- Solution of multiple subproblems or generation of multiple solutions in parallel.
- Generation of decomposition cuts for various relaxed polyhedra - diversity of cuts

**Branch-and-Relax-and-Cut**: Computational focus thus far has been on CPM/DC/PC

**General algorithmic improvements**
- Improvements to warm-starting of node solves
- Improved search strategy
- Improved branching (strong branching, pseudo-cost branching, etc.)
- Better dual stabilization
- Improved generic column generation (multiple columns generated per round, etc)

**Addition of generic MILP techniques**
- Heuristics, branching strategies, presolve
- Gomory cuts in Price-and-Cut
Current Research

- **Block structure** *(Important!)*
  - Identical subproblems for eliminating symmetry
  - Better automatic detection

- **Parallelism**
  - Parallel solution of subproblems with block structure
  - Parallelization of search using ALPS
  - Solution of multiple subproblems or generation of multiple solutions in parallel.
  - Generation of decomposition cuts for various relaxed polyhedra - diversity of cuts

- **Branch-and-Relax-and-Cut**: Computational focus thus far has been on CPM/DC/PC

- **General algorithmic improvements**
  - Improvements to warm-starting of node solves
  - Improved search strategy
  - Improved branching (strong branching, pseudo-cost branching, etc.)
  - Better dual stabilization
  - Improved generic column generation (multiple columns generated per round, etc)

- **Addition of generic MILP techniques**
  - Heuristics, branching strategies, presolve
  - Gomory cuts in Price-and-Cut
Current Research

- **Block structure (Important!)**
  - Identical subproblems for eliminating symmetry
  - Better automatic detection

- **Parallelism**
  - Parallel solution of subproblems with block structure
  - Parallelization of search using ALPS
  - Solution of multiple subproblems or generation of multiple solutions in parallel.
  - Generation of decomposition cuts for various relaxed polyhedra - diversity of cuts

- **Branch-and-Relax-and-Cut**: Computational focus thus far has been on CPM/DC/PC

- **General algorithmic improvements**
  - Improvements to warm-starting of node solves
  - Improved search strategy
  - Improved branching (strong branching, pseudo-cost branching, etc.)
  - Better dual stabilization
  - Improved generic column generation (multiple columns generated per round, etc)

- **Addition of generic MILP techniques**
  - Heuristics, branching strategies, presolve
  - **Gomory cuts** in Price-and-Cut