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

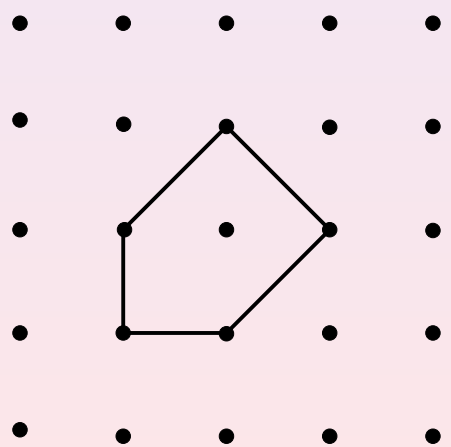
Outline

The Basic Setting

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

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

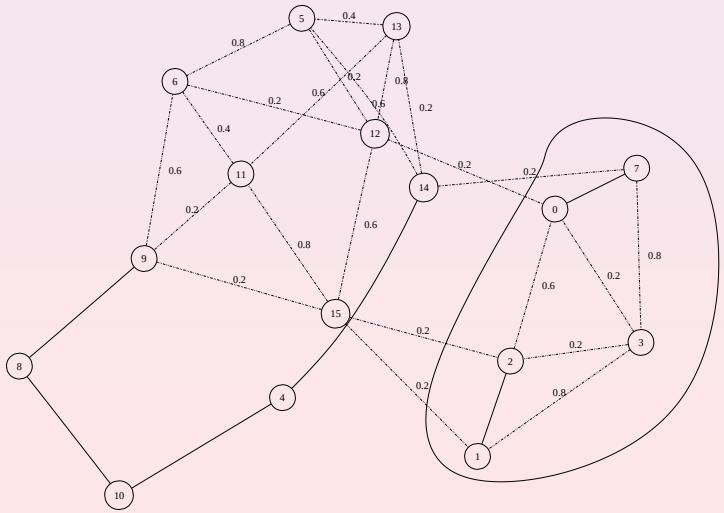
$$z_{\mathrm{LP}} = \min_{x \in \mathbb{R}^n} \left\{ c^\top x \mid A'x \geq b', A''x \geq b'' \right\}$$

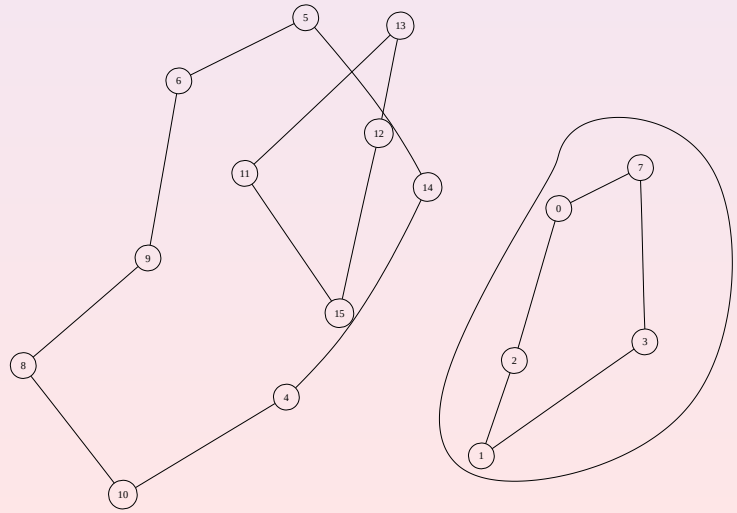


$\mathcal{P} = \mathrm{conv}\{x \in \mathbb{Z}^n \mid A'x \geq b', A''x \geq b''\}$

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

The Decomposition Principle in Integer Programming

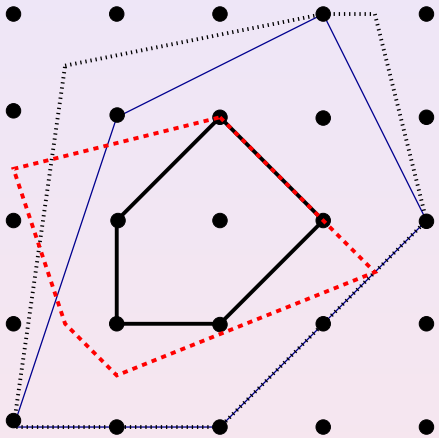
Basic Idea: By leveraging our ability to solve the optimization/separation problem for a relaxation, we can improve the bound yielded by the LP relaxation.

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

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

$$z_{\mathrm{D}} = \min_{x \in \mathcal{P}'} \left\{ c^\top x \mid A''x \geq b'' \right\}$$

$$z_{\mathrm{IP}} \geq z_{\mathrm{D}} \geq z_{\mathrm{LP}}$$



$\mathcal{P} = \mathrm{conv}\{x \in \mathbb{Z}^n \mid A'x \geq b', A''x \geq b''\}$
$\mathcal{P}' = \mathrm{conv}\{x \in \mathbb{Z}^n \mid A'x \geq b'\}$
$\mathcal{Q}' = \{x \in \mathbb{R}^n \mid A'x \geq b'\}$
$\mathcal{Q}'' = \{x \in \mathbb{R}^n \mid A''x \geq b''\}$

Assumptions:

- $\mathrm{OPT}(\mathcal{P}, c)$ and $\mathrm{SEP}(\mathcal{P}, x)$ are *"hard"*
- $\mathrm{OPT}(\mathcal{P}', c)$ and $\mathrm{SEP}(\mathcal{P}', x)$ are *"easy"*
- $\mathcal{Q}''$ 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**: $z_{\text{CP}} = \min_{x \in \mathbb{R}^n} \left\{ c^\top x \mid Dx \geq d, A''x \geq b'' \right\}$
- **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**: $z_{\text{DW}} = \min_{\lambda \in \mathbb{R}^{\mathcal{E}}_+} \left\{ c^\top \left(\sum_{s \in \mathcal{E}} s\lambda_s \right) \mid A'' \left(\sum_{s \in \mathcal{E}} s\lambda_s \right) \geq b'', \sum_{s \in \mathcal{E}} \lambda_s = 1 \right\}$
- **Subproblem**: $\text{OPT}\left(\mathcal{P}', c^\top - u_{\text{DW}}^\top A''\right)$

Lagrangian Method (LD)

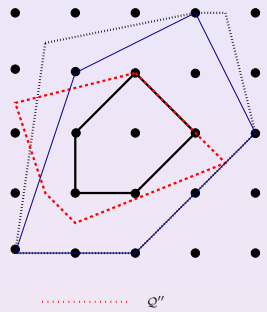
LD iteratively produces single extreme points of $\mathcal{P}'$ and uses their violation of constraints of $\mathcal{Q}''$ to converge to the same optimal face of $\mathcal{P}'$ as CPM and DW.

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

Common Threads

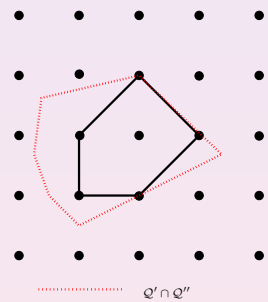
- The LP bound is obtained by optimizing over the intersection of two explicitly defined polyhedra.

$$z_{\text{LP}} = \min_{x \in \mathbb{R}^n} \{c^\top x \mid x \in \mathcal{Q}' \cap \mathcal{Q}''\}$$

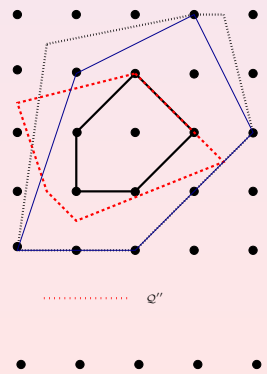


- The decomposition bound is obtained by optimizing over the intersection of one explicitly defined polyhedron and one implicitly defined polyhedron.

$$z_{\text{CP}} = z_{\text{DW}} = z_{\text{LD}} = z_{\text{D}} = \min_{x \in \mathbb{R}^n} \{c^\top x \mid x \in \mathcal{P}' \cap \mathcal{Q}''\} \geq z_{\text{LP}}$$



- Traditional decomp-based bounding methods contain two primary steps
  - **Master Problem:** Update the primal/dual **solution** information
  - **Subproblem:** Update the **approximation** of $\mathcal{P}'$: $\text{SEP}(\mathcal{P}', x)$ or $\text{OPT}(\mathcal{P}', c)$
- 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)



Generic Decomposition-based Branch and Bound

- 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 $\mathcal{P}'$);
  - 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.

Working in the Compact Space

- 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.

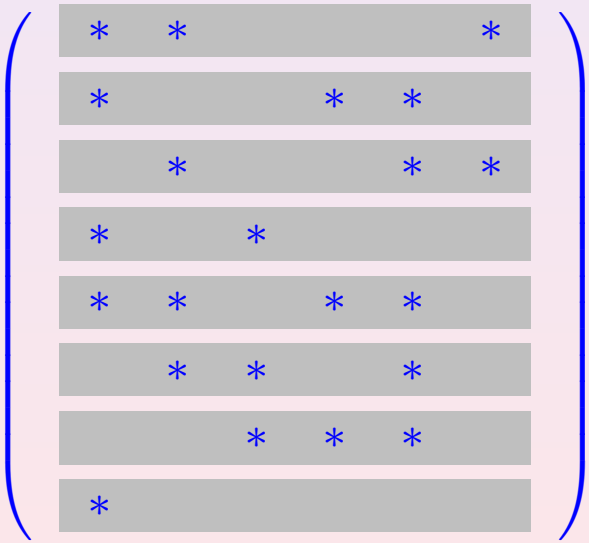
Block Structure

- 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.

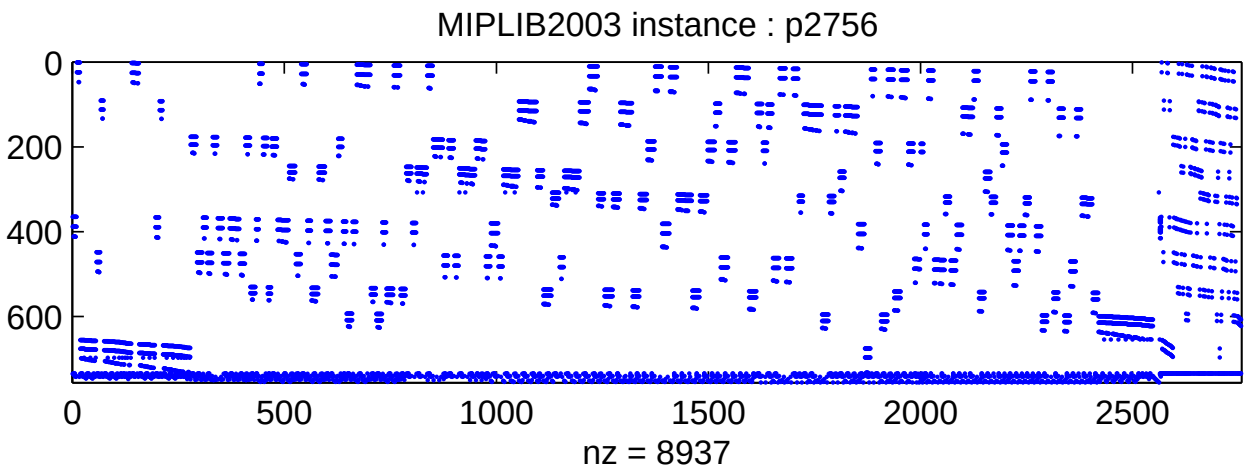
$$\begin{pmatrix} A''_1 & A''_2 & \cdots & A''_\kappa \\ \hline A'_1 & & & \\ & A'_2 & & \\ & & \ddots & \\ & & & A'_\kappa \end{pmatrix}$$

Automatic Structure Detection

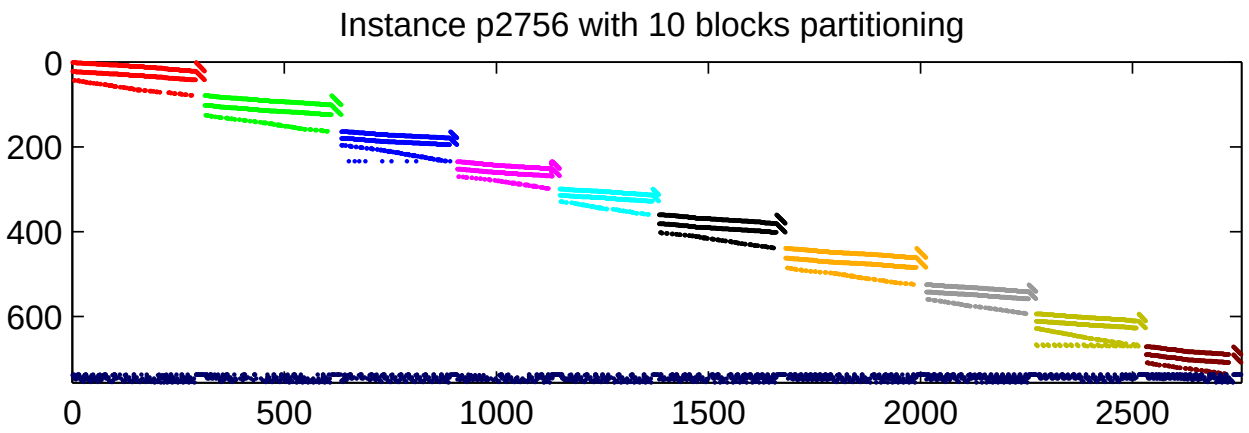
- For unstructured problems, block structure may be detected automatically.
- This is done using hypergraph partitioning methods.
- We map each row of the original matrix to a hyperedge and the nonzero elements to nodes in a hypergraph.
- Hypergraph partitioning results in identification of the blocks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



Hidden Block Structure

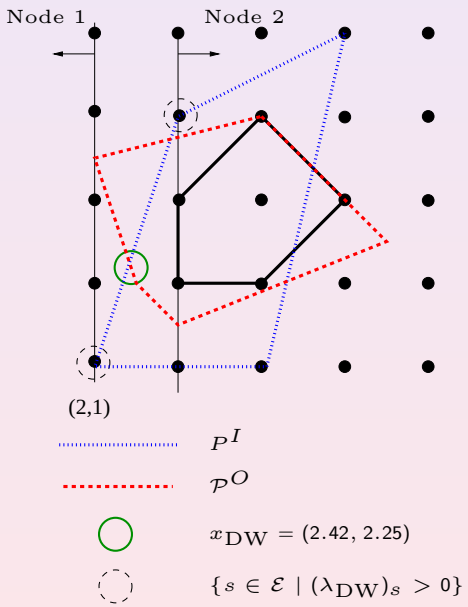


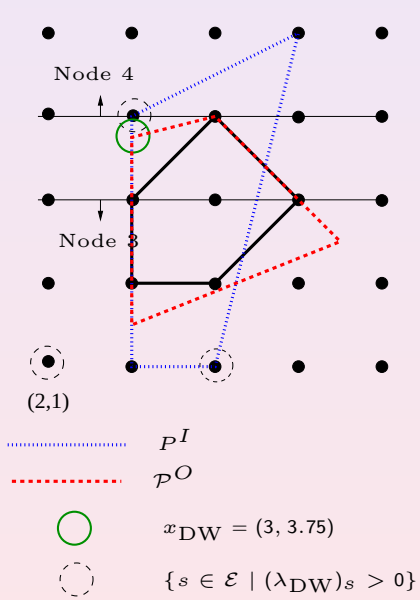
Hidden Block Structure

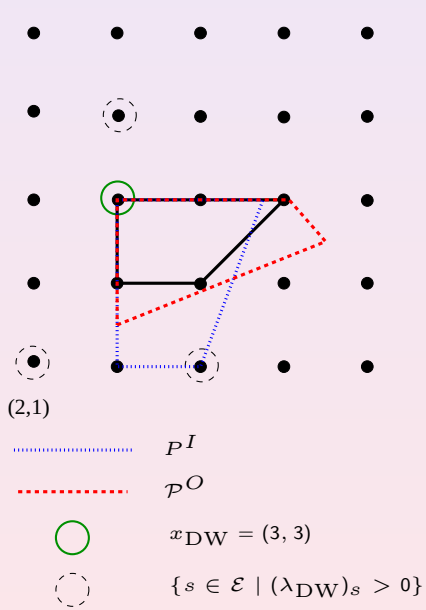


Generic Branching

- 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 define generic branching procedures.





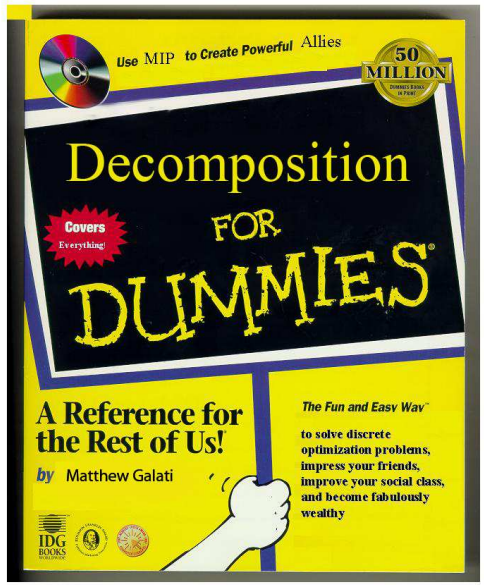


$$\begin{array}{lrcl} \text{Node 1:} & 4\lambda_{(4,1)} + 5\lambda_{(5,5)} + 2\lambda_{(2,1)} + 3\lambda_{(3,4)} & \leq & 2 \\ \text{Node 2:} & 4\lambda_{(4,1)} + 5\lambda_{(5,5)} + 2\lambda_{(2,1)} + 3\lambda_{(3,4)} & \geq & 3 \end{array}$$

Outline

DIP and CHiPPS

- 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 $\mathcal{P}'$ 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.

DIP Framework API

- 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 $\mathcal{Q}''$
    - `setModelRelaxed(DecompConstraintSet * model, int block)`: define $\mathcal{Q}'$ [optional]
  - `solveRelaxed()`: define a method for $\mathrm{OPT}(\mathcal{P}', c)$ [optional, if $\mathcal{Q}'$, **CBC** is built-in]
  - `generateCuts()`: define a method for $\mathrm{SEP}(\mathcal{P}', x)$ [optional, **CGL** is built-in]
  - `isUserFeasible()`: is $\hat{x} \in \mathcal{P}$? [optional, if $\mathcal{P} = \mathrm{conv}(\mathcal{P}' \cap \mathcal{Q}'' \cap \mathbb{Z})$]
  - 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

DipPy

- 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)

$$
\begin{aligned}
\min \quad & \sum_{i \in M} \sum_{j \in N} c_{ij} x_{ij} & & & \\
& \sum_{j \in N} w_{ij} x_{ij} & \leq & b_i & \forall i \in M \\
& \sum_{i \in M} x_{ij} & = & 1 & \forall j \in N \\
& x_{ij} & \in & \{0,1\} & \forall i,j \in M \times N
\end{aligned}
$$

GAP in DipPy

Creating GAP model in DipPy

```
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)
```

GAP in DipPy

Solver for subproblem for GAP in DipPy

```
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

```
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',
})
```

GAP in C++

DIP createModels for GAP example

```
void GAP_DecompApp::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

```
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;
}
```

GAP in C++

DIP main for GAP

```
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();
}
```

Command Line Interface

- 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

Experiments

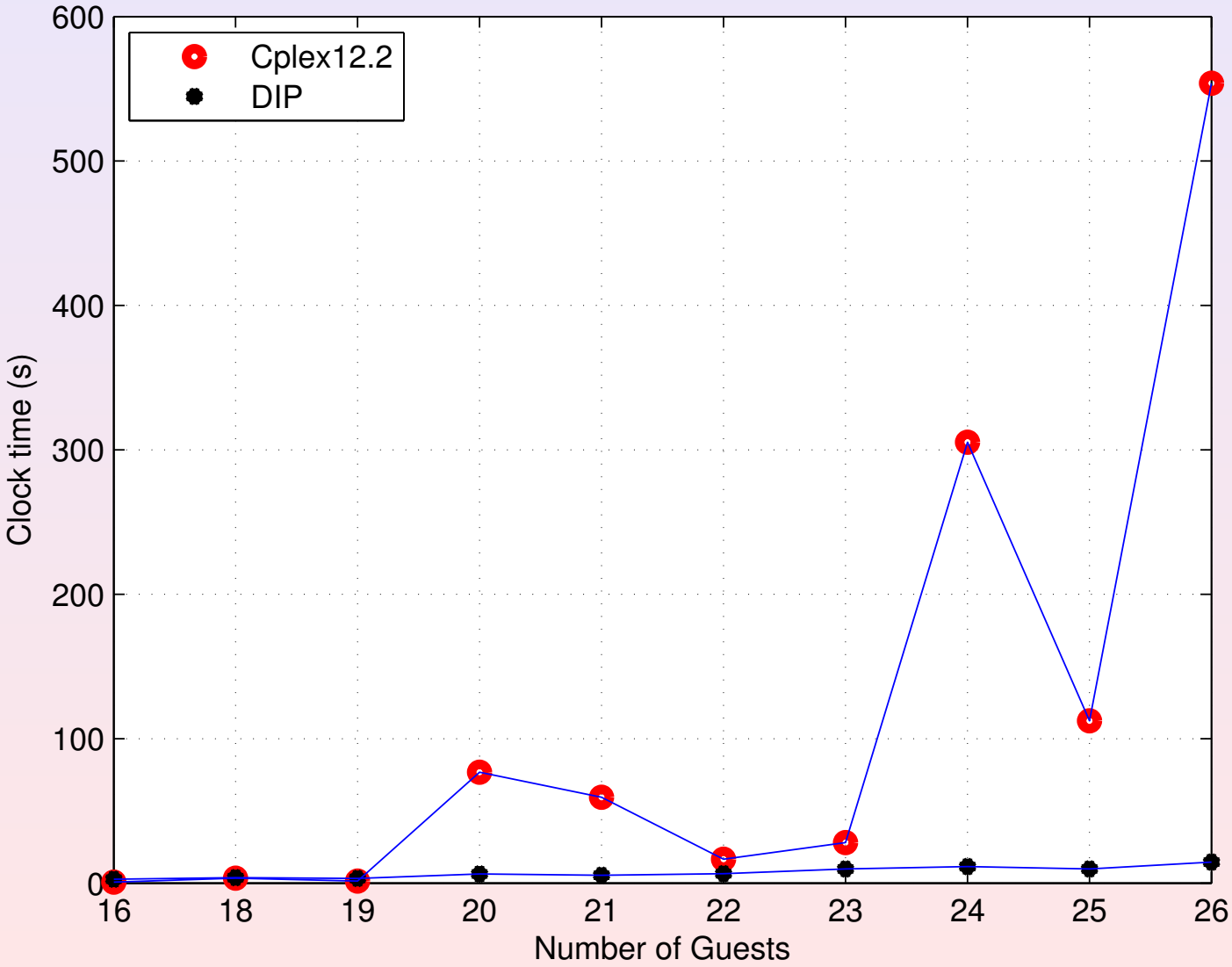
- Instances: ATM cash management, Wedding planner problem, MIPLIB
- Computational environment: single nodes of a cluster, each of which has two 8 core processors, each running at 2 GHz and with 512 KB cache, 32 GB of shared memory,

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 bock structure to parallelize the subproblem solve?

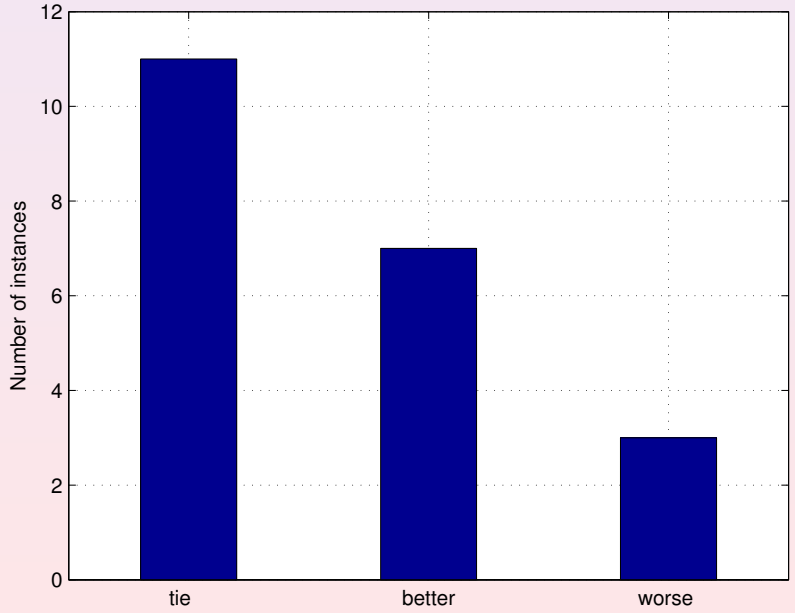
- We use bound at the root node as our primary measure for comparison in most of the following experiments.
- Caveat: these results are preliminary and their purpose is more to raise questions than answer them!

Wedding Planner Problem: CPLEX vs DIP



Comparing Hypergraph Partitioners

- 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.



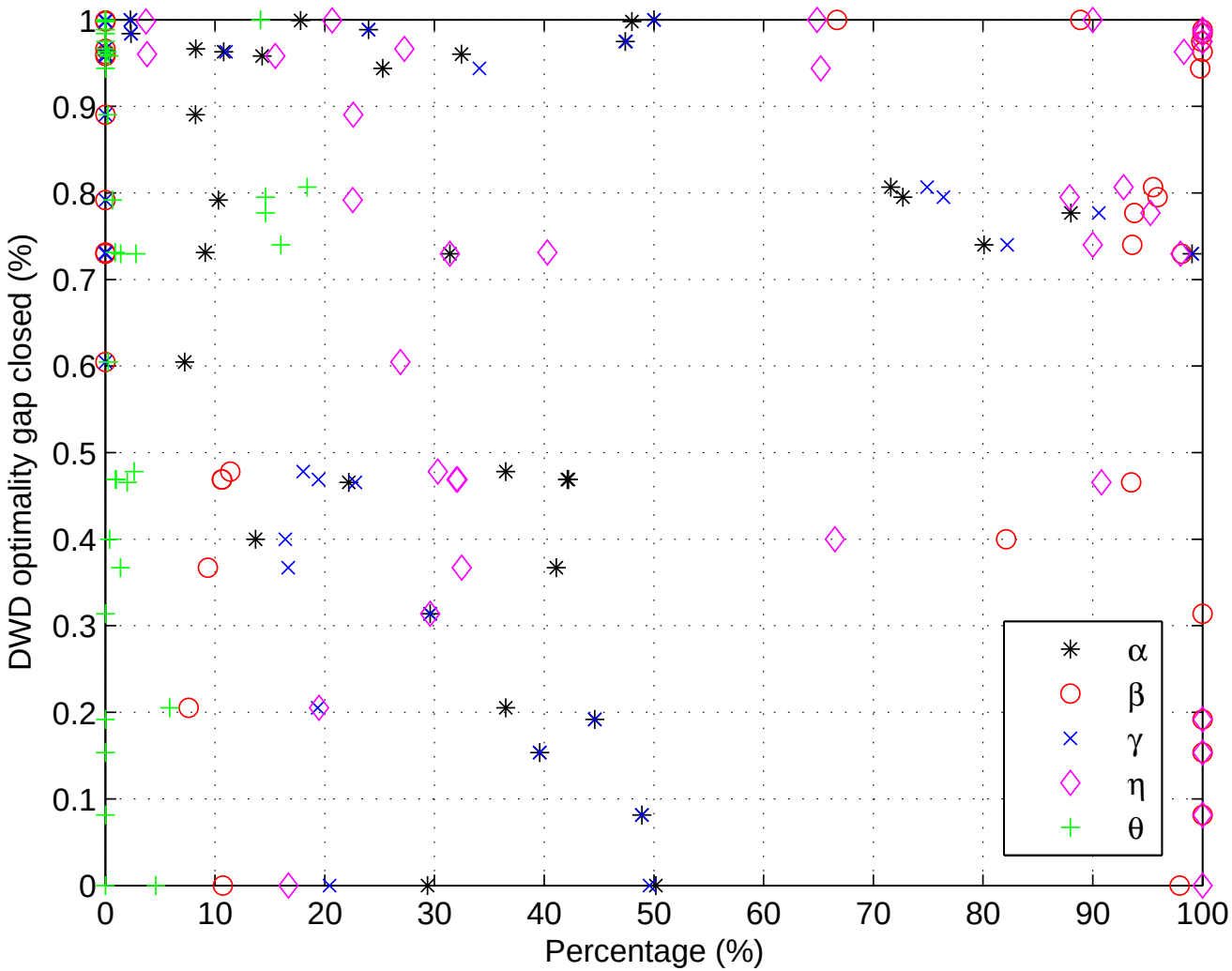
Quality Measures for Decomposition

- 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

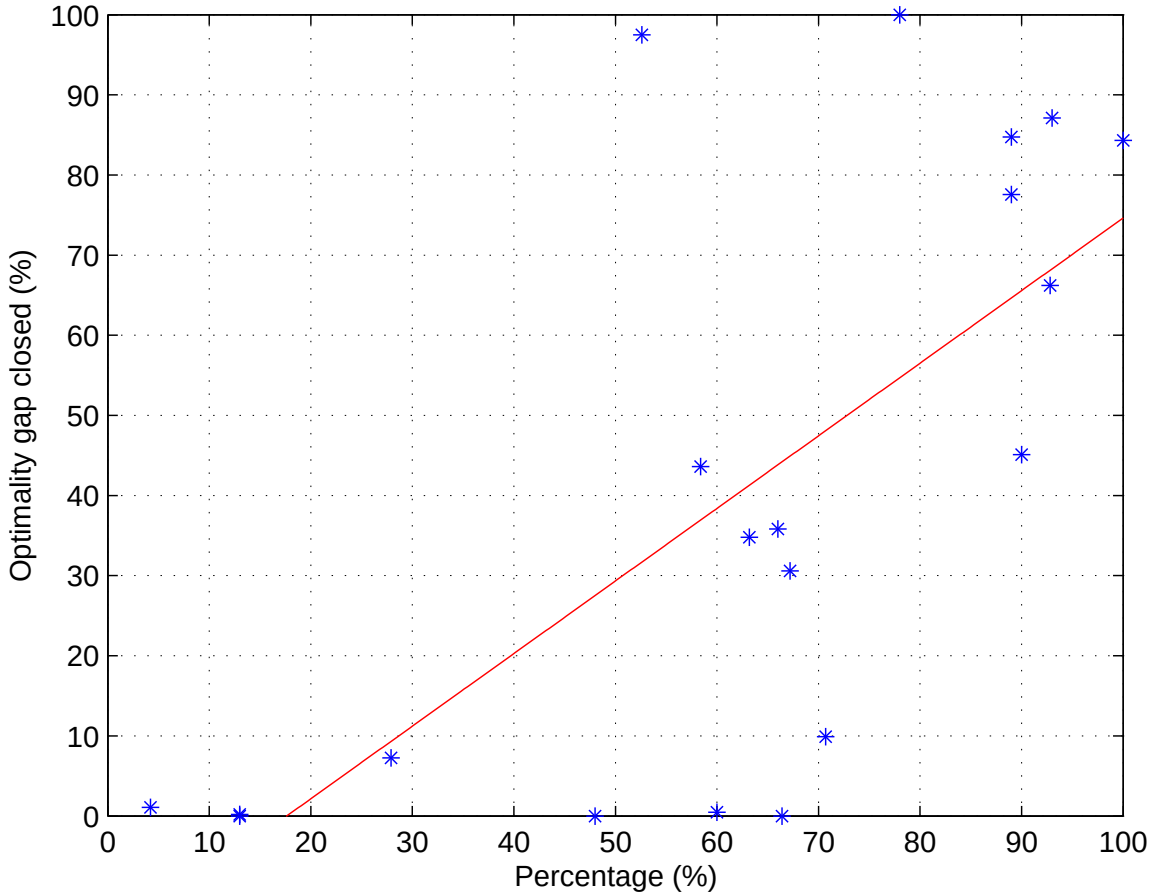
- The fraction of nonzero elements in the matrix appearing in the coupling rows (α),
- The fraction of nonzeros appearing in the coupling rows that are in integer columns (β),
- The fraction of the nonzeros in integer columns in the matrix that appear in coupling rows (γ),
- The average fraction of the nonzeros in each block that are in integer columns (η),
- The standard deviation of the fraction of integer elements elements in the blocks (θ).

Relationship between Features



A Measure for Decomposition Quality

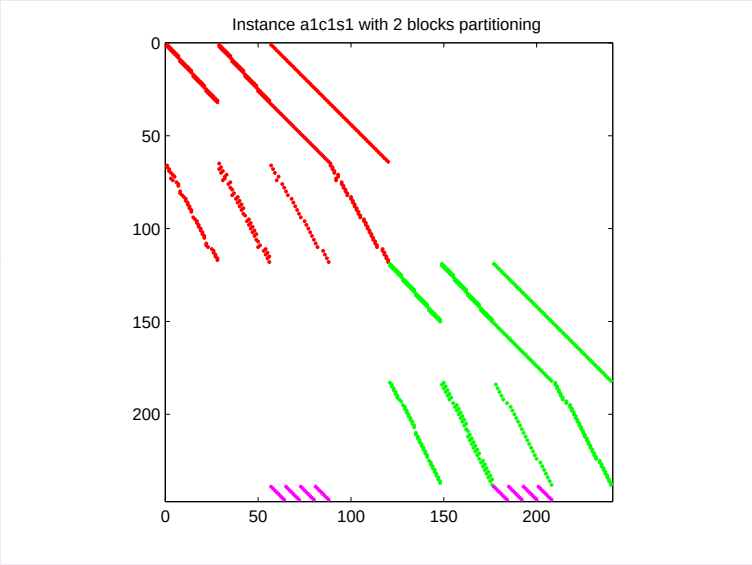
$$\Pi = (1 - \min(\alpha, \gamma))) \times 100\%,$$

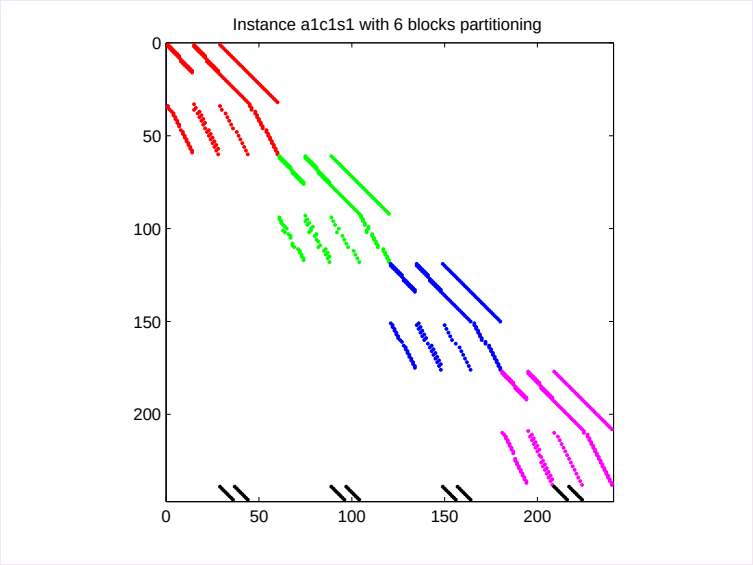


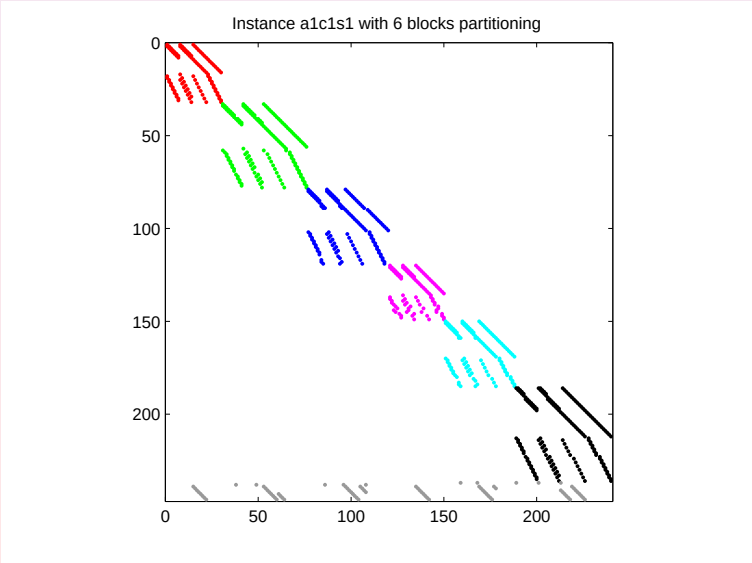
Tuning the Hypergraph Partitioning

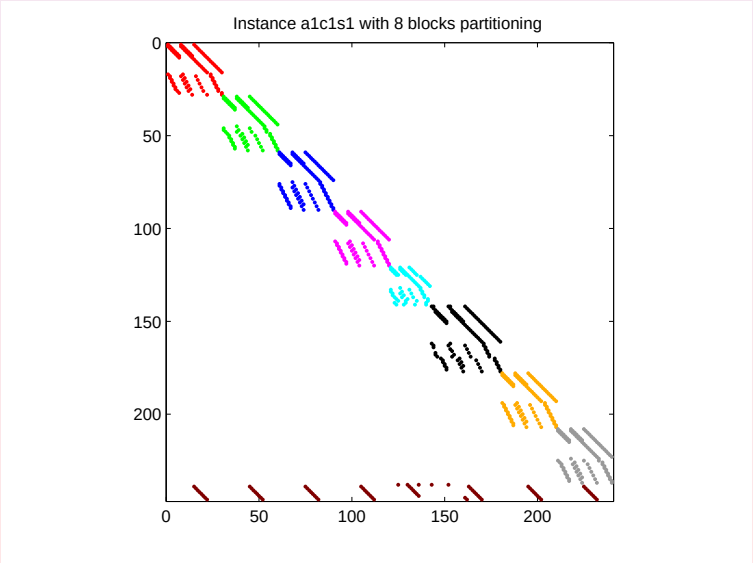
- We have now seen the features that are considered "important" in identifying a good decomposition.
- How do we encourage the partitioner to give us such a decomposition?
- With respect to the underlying graph, the partitioner has two goals.
  - The weight of the cut should be minimized.
  - The partition 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...









Effect of Number of Blocks

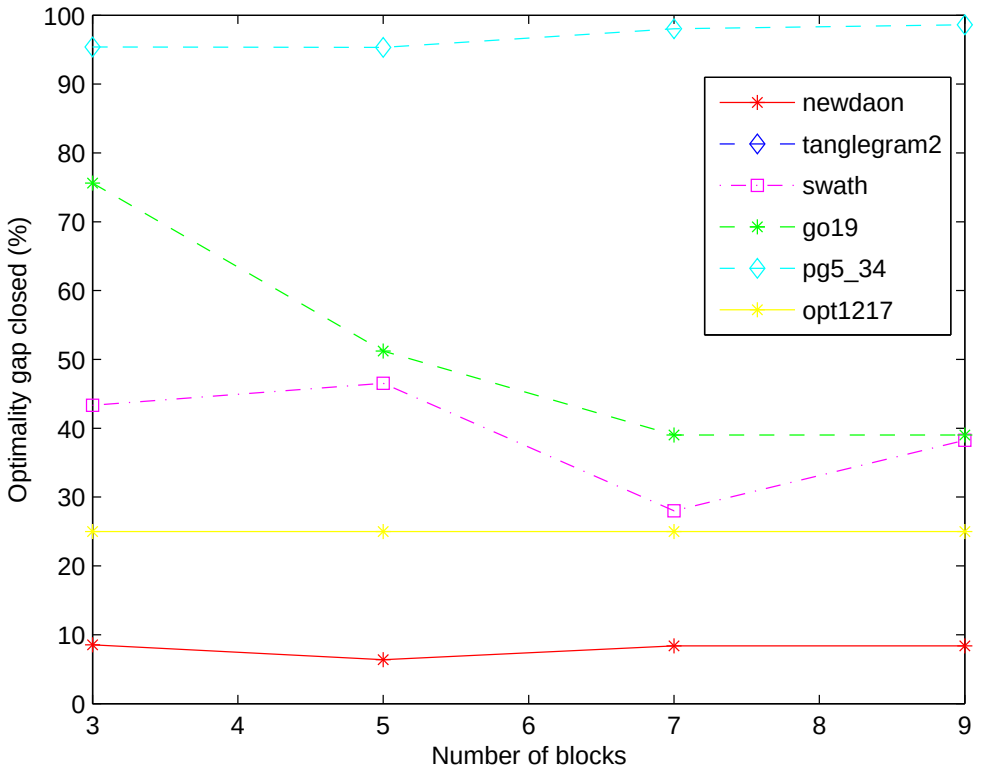
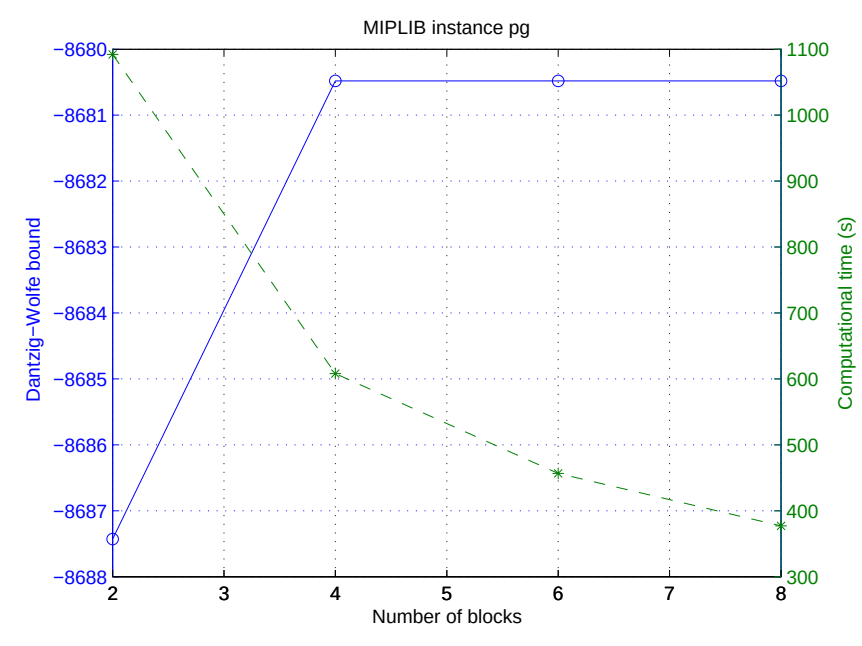
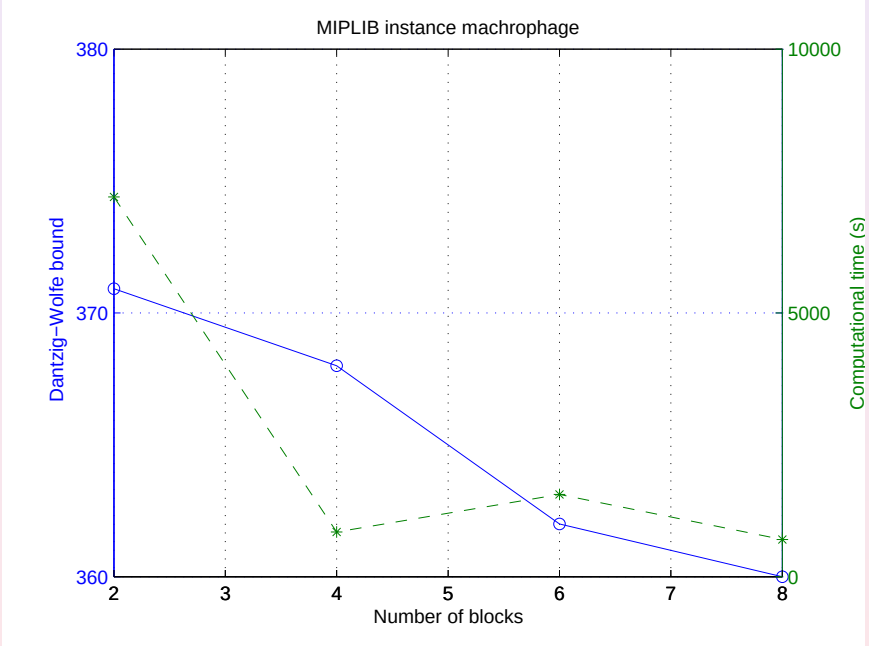


Figure: Relationship between number of blocks and optimality gap closed

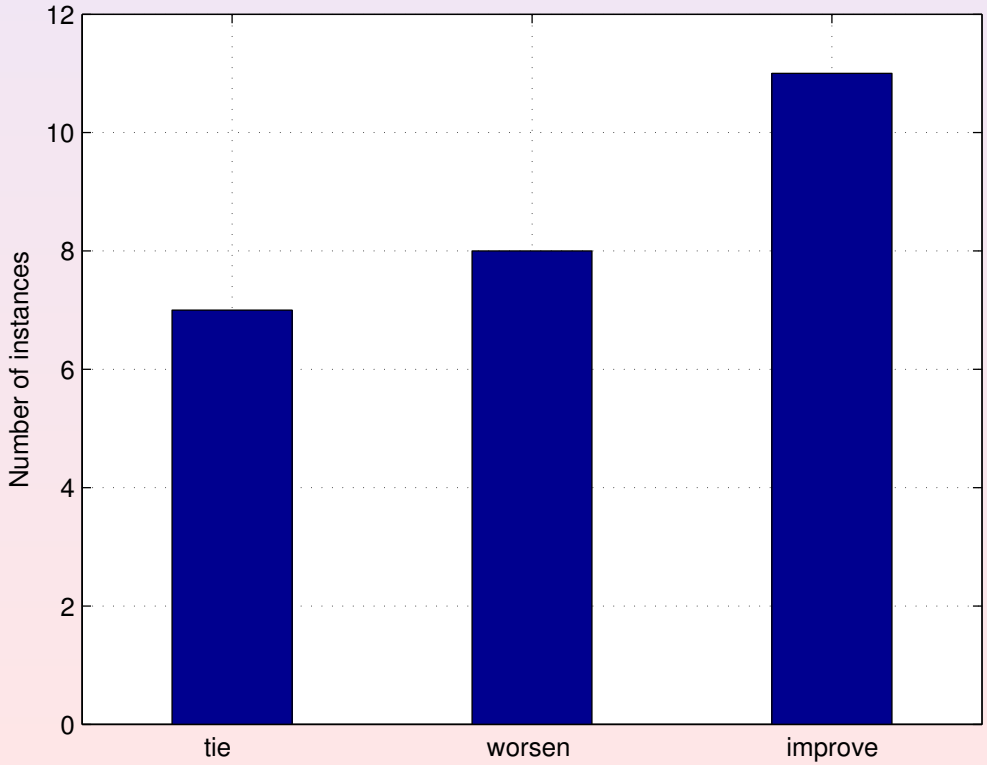
Number of Blocks, DW Bound, and Computation Time



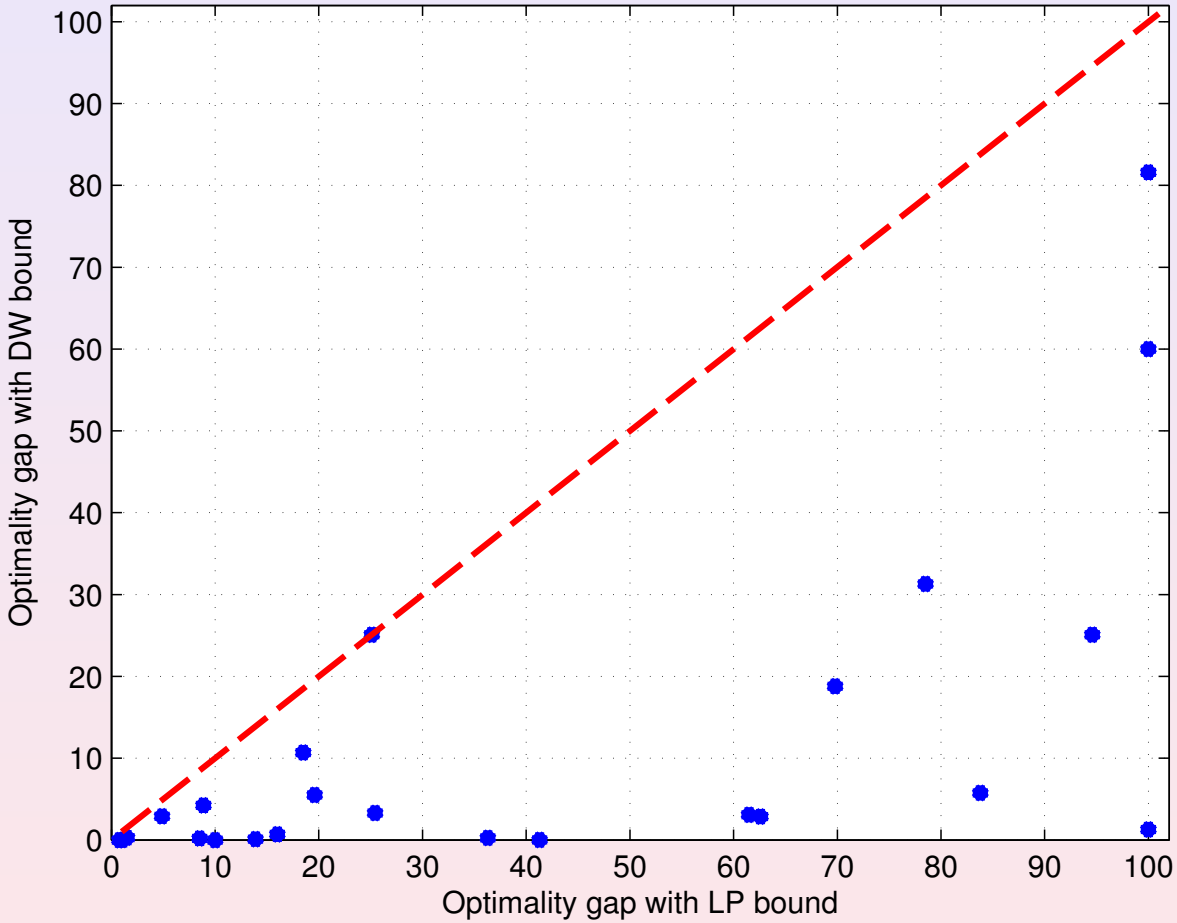


Weighting the Hyperedges and Nodes

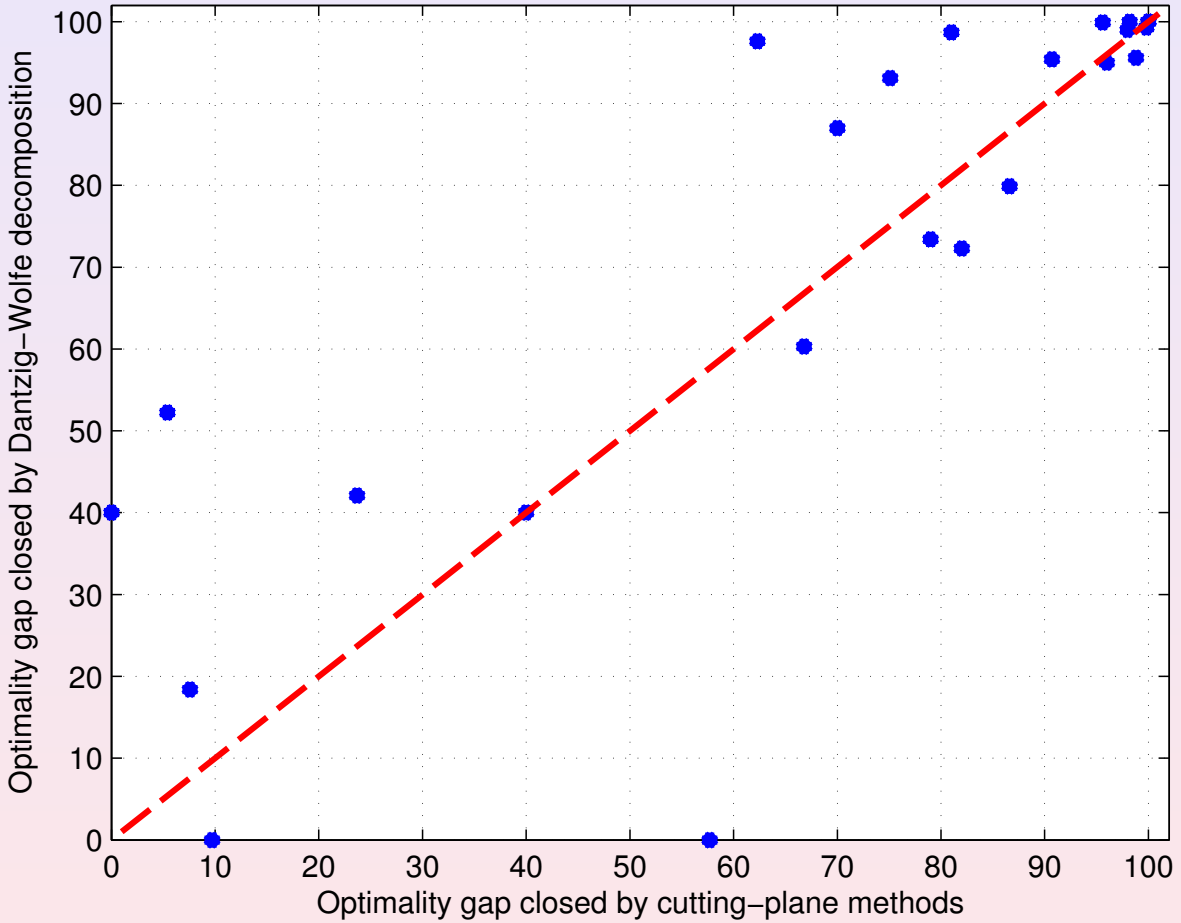
- Unit weight as input of hypergraph partitioner already gives nice results, but can we do better?
- We have already seen that it is advantageous for the rows involving more integer variables to be in the blocks.
- We assign higher weight to nodes that correspond to integer variables and to hyperedges which correspond to rows containing integer elements.



Bound at the Root Node



Optimality Gap Closed



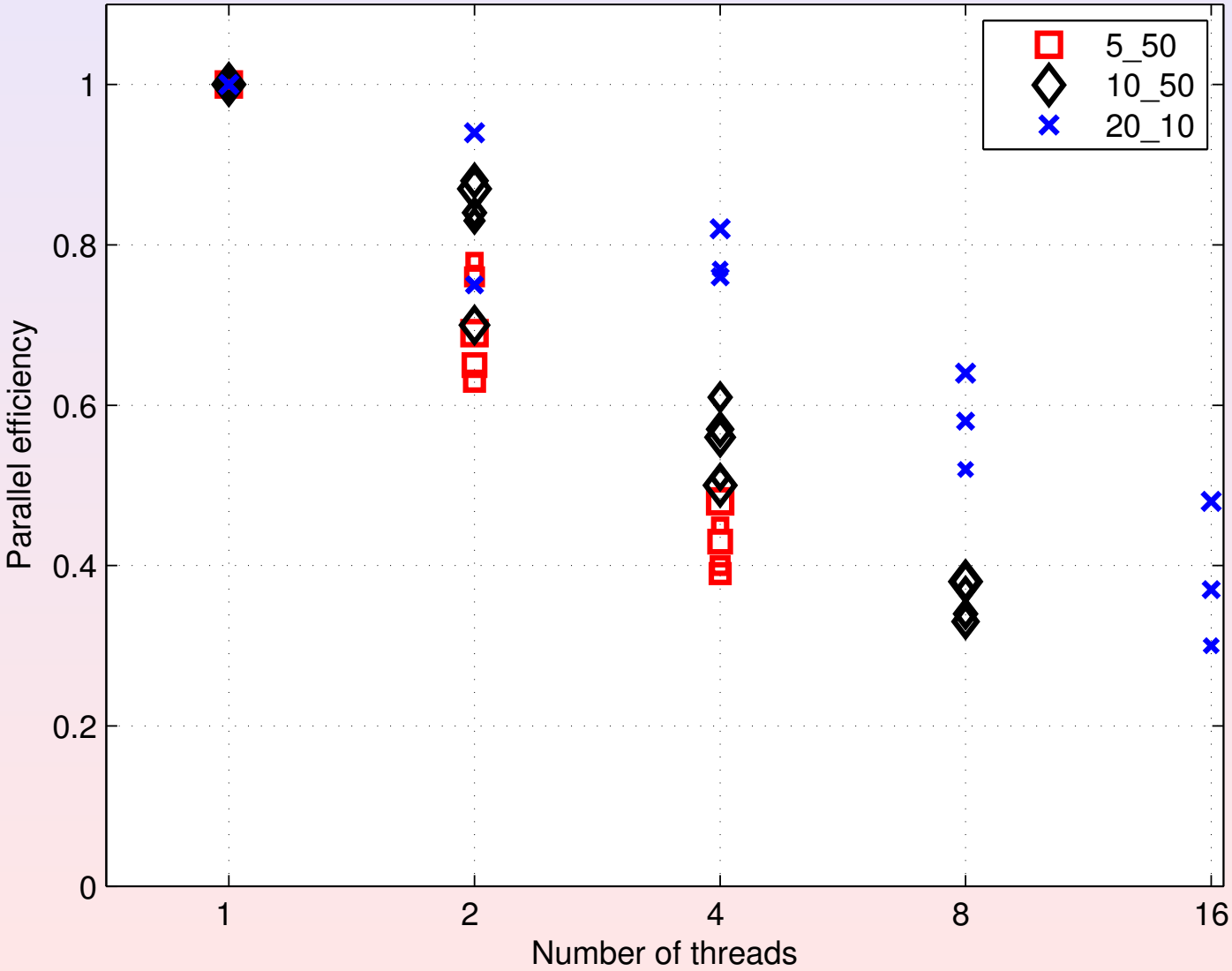
Parallelizing DIP

- 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:

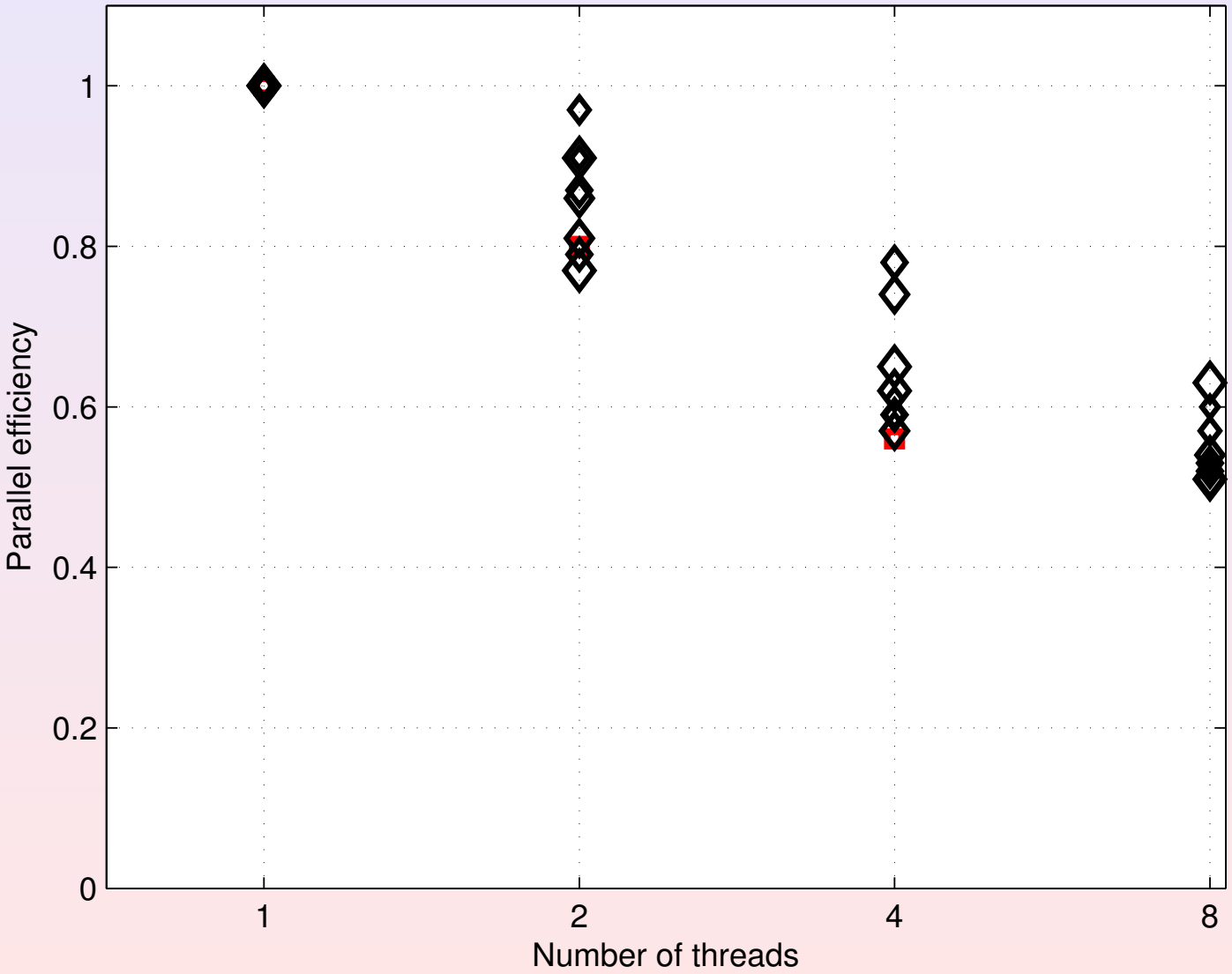
$$E = \frac{T_1}{T_p \times p},$$

where T_p is defined as the wallclock time using p threads.

ATM Problem: Efficiency vs Cores



Wedding Planner Problem: Efficiency vs Cores



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.

Multi-threaded automatic Dantzig-Wolfe Decomposition

- Using automatic hypergraph partitioning, we experimented with using DIP to solve generic (unstructured) MILPs in parallel

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

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

Outline

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