Warmer Starting for Mixed Integer Linear Programs

Ted Ralphs
Menal Guzelsoy
Ashutosh Mahajan
Svetlana Oshkai

ISE Department
COR@L Lab
Lehigh University

tkralphs@lehigh.edu

INFORMS Seattle
November 2007

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

1. Introduction

2. Algorithms
   - Definitions
   - Implementation

3. Computational Experiments
   - Combinatorial Auctions
   - Primal Heuristics
Motivation

- There are a wide range of applications in which we must repeatedly solve mixed integer linear programs.
- If the instances are only slightly different, can we do better than starting from scratch?
- In principle, warm starting techniques from LP can be generalized to do this job.

Can we make it work in practice?
Applications

Iterative Algorithms

- Bicriteria optimization algorithms
- Primal heuristics (RINS)
- Column generation algorithms for MILP
- Dual decomposition algorithm for stochastic integer programs

Real-time Optimization

- Airline Disaster Recovery
- Stochastic Vehicle Routing
- Combinatorial Auctions
Many optimization algorithms can be viewed as iterative procedures for satisfying optimality conditions (based on duality). These conditions provide a measure of “distance from optimality.” Warm starting information is additional input data that allows an algorithm to quickly get “close to optimality.” In mixed integer linear programming, the duality gap is the usual measure. A starting partition can quickly reduce the gap.

What is a starting partition and where do we get one?
Consider the implicit **optimality conditions** associated with an algorithm branch and bound.

Let $\mathcal{P}_1, \ldots, \mathcal{P}_s$ be a set of polyhedra whose union contains the feasible set.

Let $B^i$ be the optimal basis for the LP $\min_{x^i \in \mathcal{P}_i} c^\top x^i$.

Then the following is a valid lower bound

$$L = \min \{ c_{B^i}(B^i)^{-1} b + \gamma_i \mid 1 \leq i \leq s \}$$

where $\gamma_i$ is a constant factor associated with the nonbasic variables fixed at nonzero bounds.

A similar function yields an upper bound.

We call a partition that yields lower and upper bounds equal is called an **optimal partition**.
Bounding Functions

- The function
  \[ L(d) = \min \{ c_B(B^i)^{-1}d + \gamma_i \mid 1 \leq i \leq s \} \]

  provides a valid lower bound as a function of the right-hand side.

- Here is the corresponding upper bounding function
  \[ U(c) = \min \{ c_B(B^i)^{-1}b + \beta_i \mid 1 \leq i \leq s, \hat{x}^i \in \Pi \} \]

- These functions can be used for local sensitivity analysis, just as one would do in linear programming.
  - For changes in the right-hand side, the lower bound remains valid.
  - For changes in the objective function, the upper bound remains valid.
  - One can also make other modifications, such as adding variables or constraints.
To allow resolving from a warm start, we have a data structure for storing warm starts.

A warm start consists of a snapshot of the search tree, with node descriptions including:
- lists of active cuts and variables,
- branching information,
- warm start information, and
- current status (candidate, fathomed, etc.).

The tree is stored in a compact form by storing the node descriptions as differences from the parent.

Other auxiliary information is also stored, such as the current incumbent.

A warm start can be saved at any time and then reloaded later.

The warm starts can also be written to and read from disk.
Warm Starting Procedure

After modifying parameters
- If only parameters have been modified, then the candidate list is recreated and the algorithm proceeds as if left off.
- This allows parameters to be tuned as the algorithm progresses if desired.

After modifying problem data
- After modification, all leaf nodes must be modified appropriately and added to the candidate list.
- After constructing the candidate list, we can continue the algorithm as before.
There are many opportunities for improving the basic scheme.

For instance, it may not be a good idea to start the warm start from the exact tree produced during a previous solve.

Any subtree will do.

Various ad hoc procedures can be used to prune the warm start to produce a smaller tree that may be more effective.

- First $p\%$ of the nodes produced.
- All nodes above level $p \times \max$, $0 \leq p \leq 1$.
- Top $k$ levels.
SYMPHONY: Support for Warm Starting

Currently supported

- Change to objective function (no reduced cost fixing during generation of warm start).
- Change to right hand side (cuts are discarded when resolving).
- Changes to variable bounds.
- Addition of columns.

Coming soon

- Addition of constraints (easy).
- Changes to right hand side without discarding cuts (not so easy).
- Changes to objective function with reduced cost fixing (not so easy).
SYMPHONY will calculate bounds after changing the objective or right-hand side vectors.

```c
int main(int argc, char **argv)
{
    OsiSymSolverInterface si;
    si.parseCommandLine(argc, argv);
    si.loadProblem();
    si.setSymParam(OsiSymSensitivityAnalysis, true);
    si.initialSolve();
    int ind[2];
    double val[2];
    ind[0] = 4;  val[0] = 7000;
    ind[1] = 7;  val[1] = 6000;
    lb = si.getLbForNewRhs(2, ind, val);
}
```
The following example shows a simple use of warm starting to create a dynamic algorithm.

```c
int main(int argc, char **argv)
{
    OsiSymSolverInterface si;
    si.parseCommandLine(argc, argv);
    si.loadProblem();
    si.setSymParam(OsiSymFindFirstFeasible,true);
    si.setSymParam(OsiSymSearchStrategy,
                    DEPTH_FIRST_SEARCH);
    si.initialSolve();
    si.setSymParam(OsiSymFindFirstFeasible,false);
    si.setSymParam(OsiSymSearchStrategy,
                    BEST_FIRST_SEARCH);
    si.resolve();
}
```
This example shows how to warm start after problem modification.

```c
int main(int argc, char **argv)
{
    OsiSymSolverInterface si;
    CoinWarmStart ws;
    si.parseCommandLine(argc, argv);
    si.loadProblem();
    si.setSymParam(OsiSymNodeLimit, 100);
    si.initialSolve();
    ws = si.getWarmStart();
    si.resolve();
    si.setObjCoeff(0, 1);
    si.setObjCoeff(200, 150);
    si.setWarmStart(ws);
    si.resolve();
}
```
Iterative Combinatorial Auctions

- Bidders create desired packages of items and submit bids on them.
- In each round, the provisional set of winning packages are determined by maximizing revenue for round $t$.
- Constraints ensure that each item is awarded at most once.
- Additional constraints may require that a bidder win no more than one package.
The Winner Determination Problem

Set Packing Formulation

\[ \text{max} \sum_{j \in S^t} c_j x_j \]
\[ \text{s.t.} \sum_{j \in S^t} a_{ij} x_j \leq 1, \text{ for all } i \in S \]
\[ x_j \in \{0, 1\} \text{ for all } j \in S^t, \]

\( S^t = \text{set of considered packages in round } t, \)
\( c_j = \text{the bid amount of package } j, \)
\( S = \text{the set of items being auctioned,} \)
\( a_{ij} = \begin{cases} 1, & \text{if item } i \text{ is in package } j; \\ 0, & \text{otherwise.} \end{cases} \)
\( x_j = \begin{cases} 1, & \text{if package } j \text{ is a winning package;} \\ 0, & \text{otherwise.} \end{cases} \)
Alternatively, we can formulate the WDP as a set partitioning problem.

\[
\begin{align*}
\text{max} & \quad \sum_{j \in S^t} c_j x_j \\
\text{s.t.} & \quad \sum_{j \in S^t} a_{ij} x_j = 1, \text{ for all } i \in S \\
\end{align*}
\]

- In each round \( t \), all items are awarded.
- This formulation is used under the assumption of zero disposal value.
After each round, the bidders receive feedback and nonwinning bidders can
- increase their bids for the packages already in the auction, or
- create bids for new packages.

In the next round, we need to solve a modified WDP with
- updated objective coefficients,
- new columns, or
- both
Computational Experiments

- We use the cuts generated by SYMPHONY’s SPP+CUTS package
  - star cliques
  - odd holes
  - odd antiholes
- These cuts remain valid from round to round.
- We lift the star cliques (greedily) when new columns added
- Two test cases for Set Packing and Set Partitioning cases
  1. Reset WS after each iteration.
  2. Reset WS dynamically based on
     - total #of columns added
     - change in solution times
- # of bidders (5-20) - # of items(3-18)
WDP: Set Packing Formulation

Reset WS every time:

Reset WS - RCan - pt1

Relative time in ws

ws-tl-20%  ws-tl-30%  instances  ws-tl-50%  ws-tl-100%

rel-WS   rel-noWS
Reset WS dynamically:

Max reset %:
WS-20% → 8.8% (59/672)
WS-30% → 8.7% (58/668)
WS-50% → 13.2% (36/272)
WS-100% → 9.2% (25/272)
WDP: Set Packing Formulation

Never reset WS:

![Bar chart showing relative time in ws for different instances.](chart)
WDP: Set Partitioning Formulation

Reset WS every time:

Reset WS - BCan - pt1

Relative time in ws

- ws-tl-20%
- ws-tl-30%
- ws-tl-50%
- ws-tl-100%

rel-WS rel-noWS
WDP: Set Partitioning Formulation

Reset WS dynamically:

Max reset %:
- 20% → 9.3% (27/290)
- 30% → 8.9% (44/497)
- 50% → 11.4% (33/290)
- 100% → 8.9% (26/289)
Never reset the warm start:
Relaxation Induced Neighborhood Search

- An improvement heuristic by Dana, Rothberg, Pape [2005]
- Suppose $x^*$ is an incumbent solution, $x^{LP}$ is a solution of the LP relaxation at a particular node.
An improvement heuristic by Dana, Rothberg, Pape [2005]

Suppose $x^*$ is an incumbent solution, $x^{LP}$ is a solution of the LP relaxation at a particular node.

Explore some common neighborhood of $x^{LP}$ and $x^*$ for a better feasible solution.
An improvement heuristic by Dana, Rothberg, Pape [2005]

Suppose $x^*$ is an incumbent solution, $x^{LP}$ is a solution of the LP relaxation at a particular node.

Explore some common neighborhood of $x^{LP}$ and $x^*$ for a better feasible solution.

Neighborhood is defined by fixing those $x_i$ for which $x_i^{LP} = x_i^*$

It is explored by calling a MIP solver

\[
\begin{align*}
    x^{LP} &= (0, 0, 1, 0, 0.5, 0.2, 1, 0.2, 0) \\
    x^* &= (1, 0, 0, 1, 1, 0.0, 1.0, 1, 1.0, 1)
\end{align*}
\]

$\quad x \in P(\text{original problem})$

$\quad x_1 = 0$

$\quad x_3 = 1$

$\quad x_7 = 1$

[SOLVE AS A MIP]
Warm Starting in RINS

Motivation

- If RINS is being called repeatedly, the sub-MILPs being solved will be similar.
- Perhaps warm starting can help.

Experimental setup

- We initially save the root node (LP+Cuts) as a warm start.
- Each time RINS is called:
  - Load the current warm start environment.
  - Fix $x_i$ if $x_i^{LP} = x_i^*$ by changing bounds.
  - Solve with time limit.
Computational experiments

- All mixed binary instances in miplib3(50), miplib2003(45), mittleman-unibo-instances(31).
- Compared 6 runs: Default RINS, warm-starting with node-level-ratio 0.1, 0.2, 0.5, 0.7, 1.0
- Total time limit = 2hrs. (4 hrs for mittleman-unibo)
- Time limit for each RINS call = 100s (300s for mittleman-unibo)

**Average behavior over all instances**

<table>
<thead>
<tr>
<th></th>
<th>Def</th>
<th>nl-ratio=0.1</th>
<th>0.2</th>
<th>0.5</th>
<th>0.7</th>
<th>1.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avg. Time-per-call</td>
<td>1.0</td>
<td>2.53</td>
<td>2.49</td>
<td>2.46</td>
<td>2.48</td>
<td>2.59</td>
</tr>
<tr>
<td>Avg. Def</td>
<td>158</td>
<td>199</td>
<td>190</td>
<td>184</td>
<td>215</td>
<td>210</td>
</tr>
</tbody>
</table>

How does the performance vary over the instances?
MIPLIB Instances: Performance Profiles

- Time spent in RINS is reduced dramatically
- Minor improvements in running time and optimality gap
- Not unexpected as only upper-bounds are affected
Results: Particular instances

- **Time to feasible solutions: mkc**
  - No-ws
  - Node level ratio 0.1
  - Node level ratio 0.2
  - Node level ratio 0.5
  - Node level ratio 0.7
  - Node level ratio 1.0

- **Time to feasible solutions: sp98ar**
  - No-ws
  - Node level ratio 0.1
  - Node level ratio 0.2
  - Node level ratio 0.5
  - Node level ratio 0.7
  - Node level ratio 1.0

- **Time to feasible solutions: aflow30a**
  - No-ws
  - Node level ratio 0.1
  - Node level ratio 0.2
  - Node level ratio 0.5
  - Node level ratio 0.7
  - Node level ratio 1.0

- **Time to feasible solutions: swath**
  - No-ws
  - Node level ratio 0.1
  - Node level ratio 0.2
  - Node level ratio 0.5
  - Node level ratio 0.7
  - Node level ratio 1.0
Combinatorial Auctions
- For combinatorial auctions, using warm starting sped up computations consistently.
- It was necessary to start from scratch periodically in order to avoid a loss of efficiency.

RINS
- Using warm start improved time spent by a factor of 2-3 times for medium and (some) large instances.
- Overall impact was low, however.
- For very large instances, carrying over a single warm-start environment does not seem to be a good idea.
- Starting from scratch if several calls to heuristic are unsuccessful might be a good idea.
- Not having cuts and reduced cost fixing is a handicap.

In both cases, it was difficult to know how to prune the warm start intelligently.

The bottom line: when solving modified instances of these classes of MILPs, warm starting does seem to be a good idea.