An Open Source Solver for Bicriteria Mixed Integer Programs

(and a few other things...)

Ted Ralphs and Menal Guzelsoy
Industrial and Systems Engineering
Lehigh University

Matthew Saltzman
Clemson University

Margaret Wiecek
Clemson University

CORS/INFORMS Joint Int'l Meeting, Banff, Alberta, Canada, Sunday, May 16, 2004

Outline of Talk

- Introduction to SYMPHONY 5.0
  - Callable library API
  - OSI interface
  - User callbacks
- Implementation
  - Solve
  - Resolve
  - Bicriteria solve
- Examples

Brief Introduction to SYMPHONY

- Overview
  - A callable library for solving mixed-integer linear programs with a wide variety of customization options.
  - Core solution methodology is a state of the art implementation of branch, cut, and price.
  - Outfitted as a generic MILP solver.
  - Fully integrated with the Computational Infrastructure for Operations Research (COIN-OR) libraries.
  - Extensive documentation available.
  - Source can be downloaded from `www.branchandcut.org`
- SYMPHONY Solvers
  - Generic MILP
  - Traveling Salesman Problem
  - Vehicle Routing Problem
  - Mixed Postman Problem
  - Set Partitioning Problem
  - Matching Problem
  - Network Routing

What is COIN-OR?

- The COIN-OR Project
  - An initiative promoting the development and use of interoperable, open-source software for operations research.
  - A consortium of researchers in both industry and academia dedicated to improving the state of computational research in OR.
  - A non-profit corporation known as the COIN-OR Foundation
- The COIN-OR Repository
  - A library of interoperable software tools for building optimization codes, as well as some stand-alone packages.
  - A venue for peer review of OR software tools.
  - A development platform for open source projects, including a CVS repository.
  - Soon to be hosted by INFORMS.

Supported Formats and Architectures

- Input formats
  - MPS (COIN-OR parser)
  - GMPL/AMPL (GLPK parser)
  - User defined
- Output/Display formats
  - Text
  - IGD
  - VbcTool
- Supported architectures
  - Single-processor Linux, Unix, or Windows
  - Distributed memory parallel (message-passing)
  - Shared memory parallel (OpenMP)

SYMPHONY C Callable Library

- Primary subroutines
  - `sym_open_environment()`
  - `sym_parse_command_line()`
  - `sym_load_problem()`
  - `sym_find_initial_bounds()`
  - `sym_solve()`
  - `sym_mc_solve()`
  - `sym_resolve()`
  - `sym_close_environment()`
- Auxiliary subroutines
  - Accessing and modifying problem data
  - Accessing and modifying parameters
  - User callbacks

Implementing a MILP Solver with SYMPHONY

- Using the callable library, we only need a few lines to implement a solver.
- The file name and other parameters are specified on the command line.
- The code is the same for any configuration or architecture, sequential or parallel.
- The code is exactly the same for all architectures, even parallel.
- Command line would be

```
symphony -F model.mps
```

```
int main(int argc, char **argv)
{
   sym_environment *p = sym_open_environment();
   sym_parse_command_line(p, argc, argv);
   sym_load_problem(p);
   sym_solve(p);
   sym_close_environment(p);
}
```

OSI interface

- The COIN-OR Open Solver Interface is a standard C++ class for accessing solvers for mathematical programs.
- Each solver has its own derived class that translates OSI calls into those of the solver's library.
- For each method in OSI, SYMPHONY has a corresponding method.
- The OSI interface is implemented as wrapped C calls.
- The constructor calls `sym_open_environment()` and the destructor calls `sym_close_environment()`.
- The OSI `initialSolve()` method calls `sym_solve()`.
- The OSI `resolve()` method calls `sym_resolve()`.

Using the SYMPHONY OSI interface

- Here is the implementation of a simple solver using the SYMPHONY OSI interface.

```
int main(int argc, char **argv)
{
   OsiSymSolverInterface si;
   si.parseCommandLine(argc, argv);
   si.loadProblem();
   si.branchAndBound();
}
```

- Again, the code is the same for any configuration or architecture, sequential or parallel.

Customizing

- The main avenues for advanced customization are the parameters and the user callback subroutines.
- There are more than 50 callbacks and over 100 parameters.
- The user can override SYMPHONY's default behavior in a variety of ways.
  - Custom input
  - Custom displays
  - Branching
  - Cut/column generation
  - Cut pool management
  - Search and diving strategies
  - LP management

Implementation of `initialSolve()`

- To enable parallelization, SYMPHONY is made up of five modules encapsulating the main tasks of branch, cut, and price.
  - Master
  - Tree Manager
  - Node Processing
  - Cut Generator
  - Cut Pool
- The master module is the only persistent entity.
- All other modules are transient and exist only when a solve call is active.
- After solve is called, the master creates the tree manager, which in turn creates the other modules.
- In parallel, these are spawned as remote processes.
- The tree manager manages the solution process and hands the results back to the master module.

Warm Starts for MILP

- To allow resolving from a warm start, we have defined a SYMPHONY warm start class, which is derived from `CoinWarmStart`.
- The class stores 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
  - We limit modifications to those that do not invalidate the node warm start information.
  - Currently, we only allow modification of rim vectors.
  - After modification, all leaf nodes must be added to the candidate list.
  - After constructing the candidate list, we can continue the algorithm as before.

Warm Starting Example (Parameter Modification)

- The following example shows a simple use of warm starting to create a dynamic algorithm.

```
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();
}
```

Warm Starting Example (Problem Modification)

- The following example shows how to warm start after problem modification.

```
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();
}
```

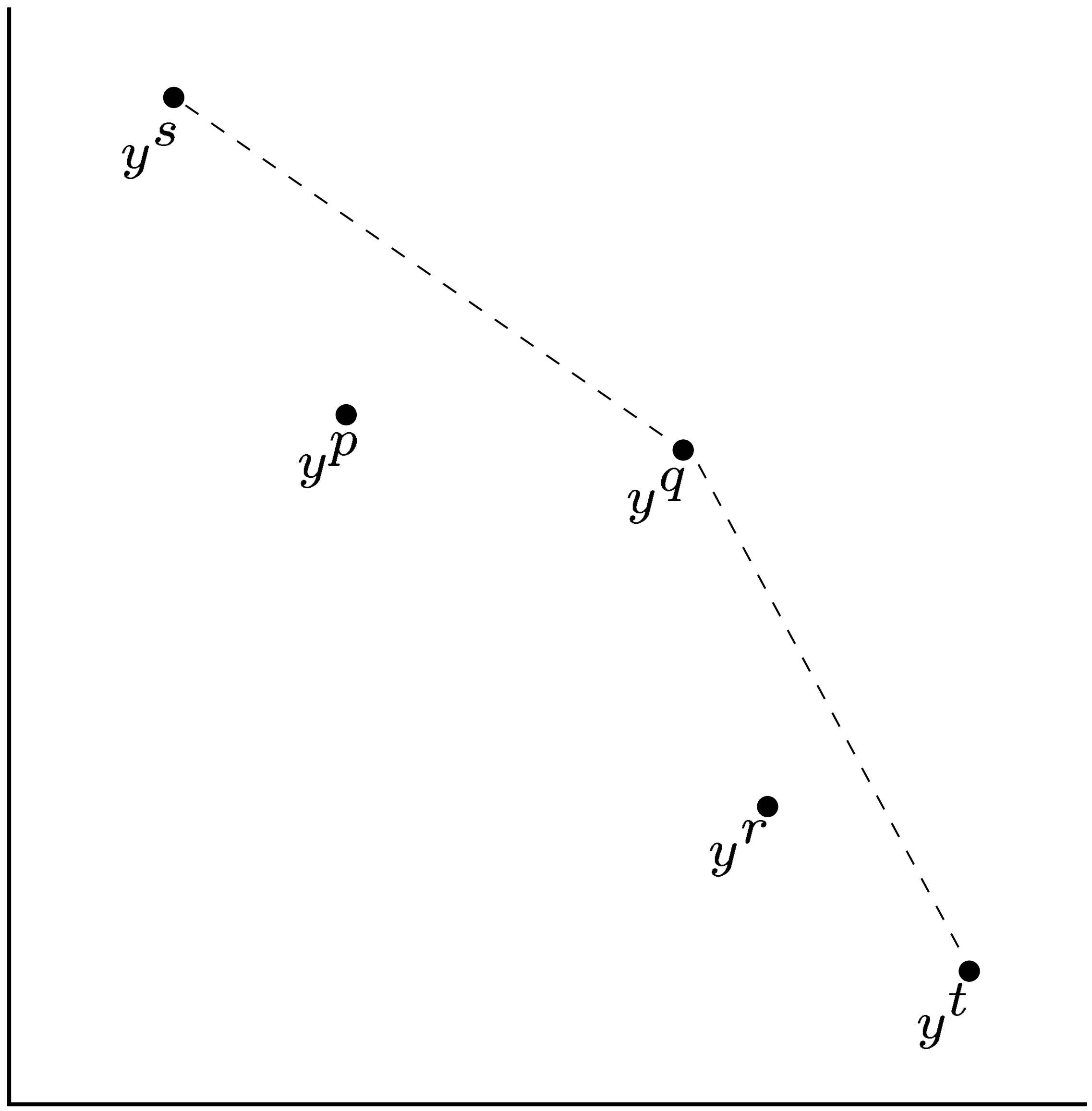
Bicriteria MILPs

- We limit the discussion here to pure integer programs (ILPs), but generalization to MILPs is straightforward.
- The general form of a bicriteria ILP is

$$\begin{aligned} \text{vmax}\ & [cx, dx], \\ \text{s.t.}\quad & Ax \le b, \\ & x \in \mathbb{Z}^n. \end{aligned}$$

- Solutions don't have single objective function values, but pairs of values called *outcomes*.
- A feasible $\hat{x}$ is called *efficient* if there is no feasible $\bar{x}$ such that $c\bar{x} \ge c\hat{x}$ and $d\bar{x} \ge d\hat{x}$, with at least one inequality strict.
- The outcome corresponding to an efficient solution is called *Pareto*.
- The goal is to enumerate Pareto outcomes.

Illustration of Pareto and Supported Outcomes



Supported Outcomes

- A bicriteria ILP can be converted to a single-criteria ILP by substituting a *weighted sum objective*

$$\max_{x \in X} (\beta c + (1 - \beta) d) x$$

for the bicriteria objective to obtain a parameterized family of ILPs.

- Optimal solutions to members of this family are extreme points of the convex lower envelope of outcomes and are called *supported*.
- Supported outcomes are Pareto, but the converse is not true.
- It is straightforward to generate all *supported outcomes* by solving a sequence of ILPs.

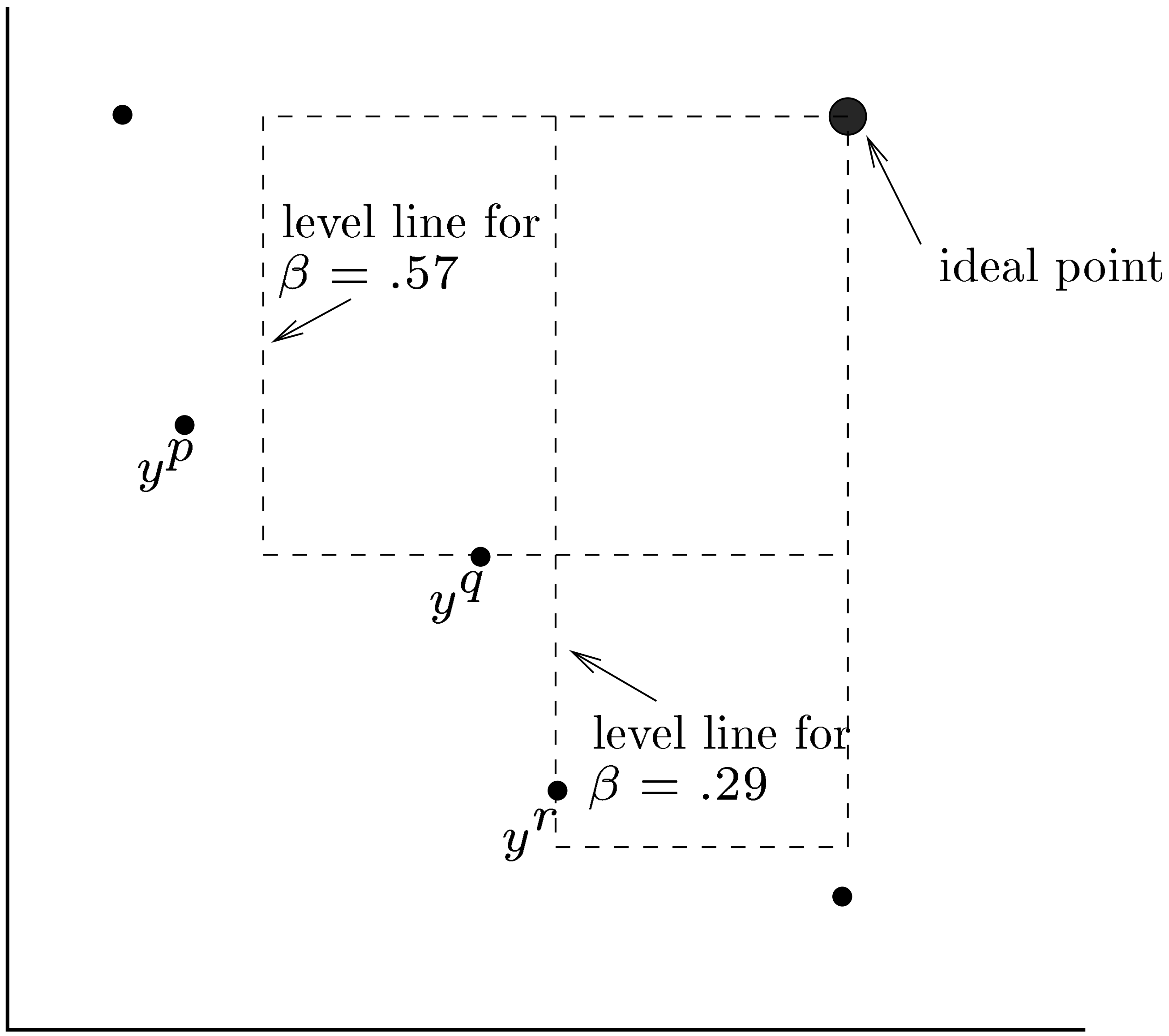
Generating Pareto Outcomes

- To generate Pareto outcomes, we must replace the weighted sum objective with a *weighted Chebyshev norm* (WCN) objective.
- Let x^c be a solution to the original ILP with objective c and x^d be a solution with objective d.
- Then the WCN objective is

$$\min_{x \in X} \max\{\beta(cx - cx^c), (1-\beta)(dx - dx^d)\}.$$

- This objective can be linearized to obtain another family of ILPs.
- Assuming *uniform dominance*, Bowman showed solutions are efficient if and only if they optimal for some member of this family.
- The mild condition is *uniform dominance*, which states that
- This family has also been studied by Eswaran and Solanki.
- In a recent paper, we described an improved algorithm for finding all Pareto outcomes by solving a sequence of ILPs from this family.

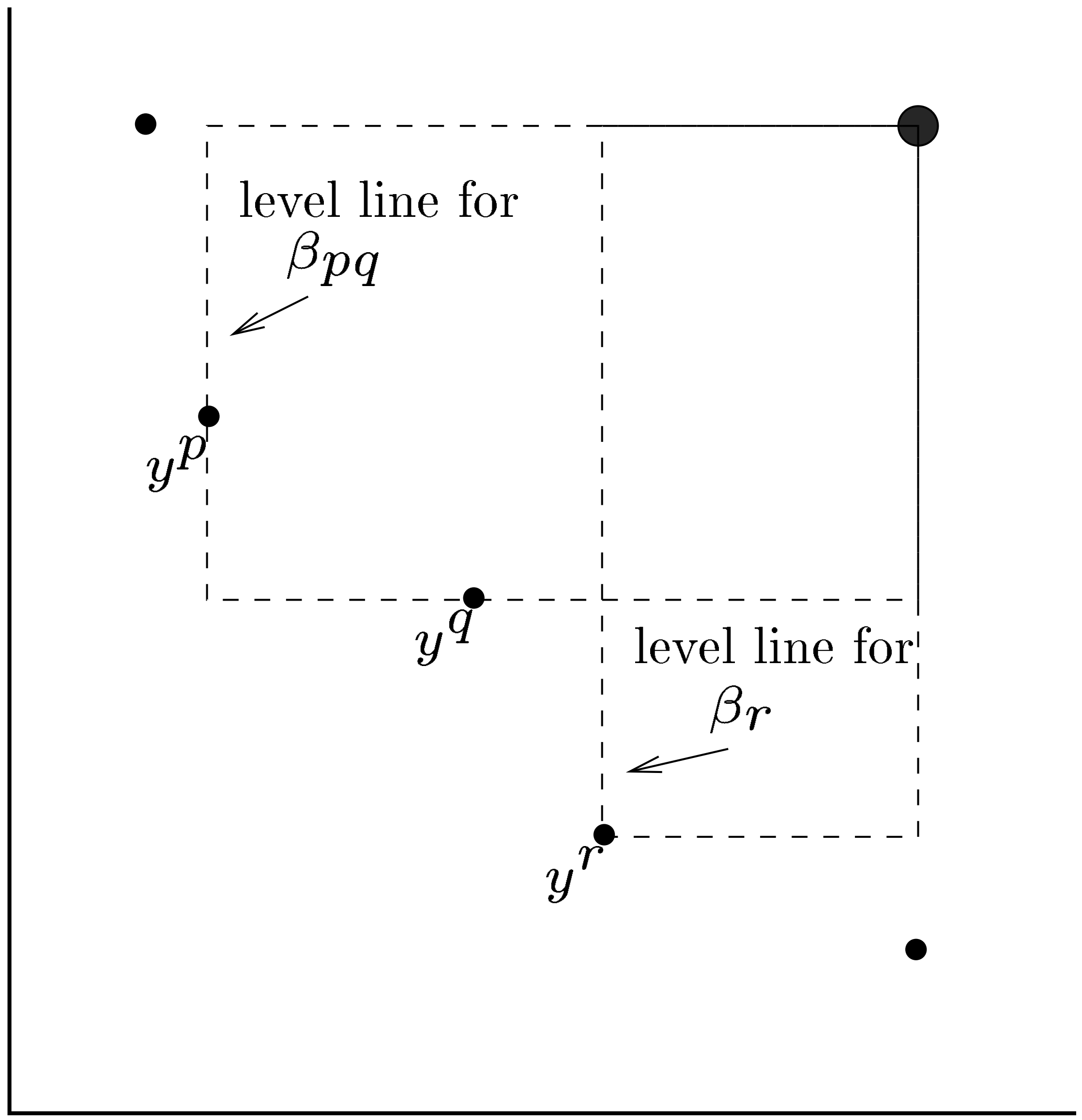
Illustration of the Chebyshev Norm



The WCN algorithm

- The algorithm maintains a list of Pareto outcomes found so far, ordered by corresponding β value.
- We choose a pair (p, q) from the list and determine whether there is a Pareto outcome between them by solving a ILP with WCN objective and weight
$$\beta_{pq} = (dx - dx^d)/(cy - cy^c + dx - dx^d),$$
- If the result is a known outcome, then β_{pq} is a breakpoint.
- Otherwise, the result is a new efficient solution r and we add (p, r) and (r, q) to the list.
- This algorithm is asymptotically optimal.

Illustration of Weights in the WCN Algorithm



Implementing the WCN algorithm

- Because the WCN algorithm involves solving a sequence of slightly modified MILPs, warm starting can be used.
- Two approaches
  - Warm start from the result of the previous iteration.
  - Solve a "base" problem first and warm each subsequent problem from there.
- In addition, we can optionally save the global cut pool from iteration to iteration, using SYMPHONY's persistent cut pools.
- If the uniform dominance assumption is not satisfied, then we have to filter out weakly dominated solutions.
- Both the callable library and the OSI interface allow the user to define a second objective function and call the bicriteria solver.

Example: Warm Starting

- Consider the simple warm-starting code from earlier in the talk.
- Applying this code to the MIPLIB 3 problem p0201, we obtain the results below.
- Note that the warm start doesn't reduce the number of nodes generated, but does reduce the solve time dramatically.

	CPU Time	Tree Nodes
Generate warm start	28	100
Solve orig problem (from warm start)	3	118
Solve mod problem (from scratch)	24	122
Solve mod problem (from warm start)	6	198

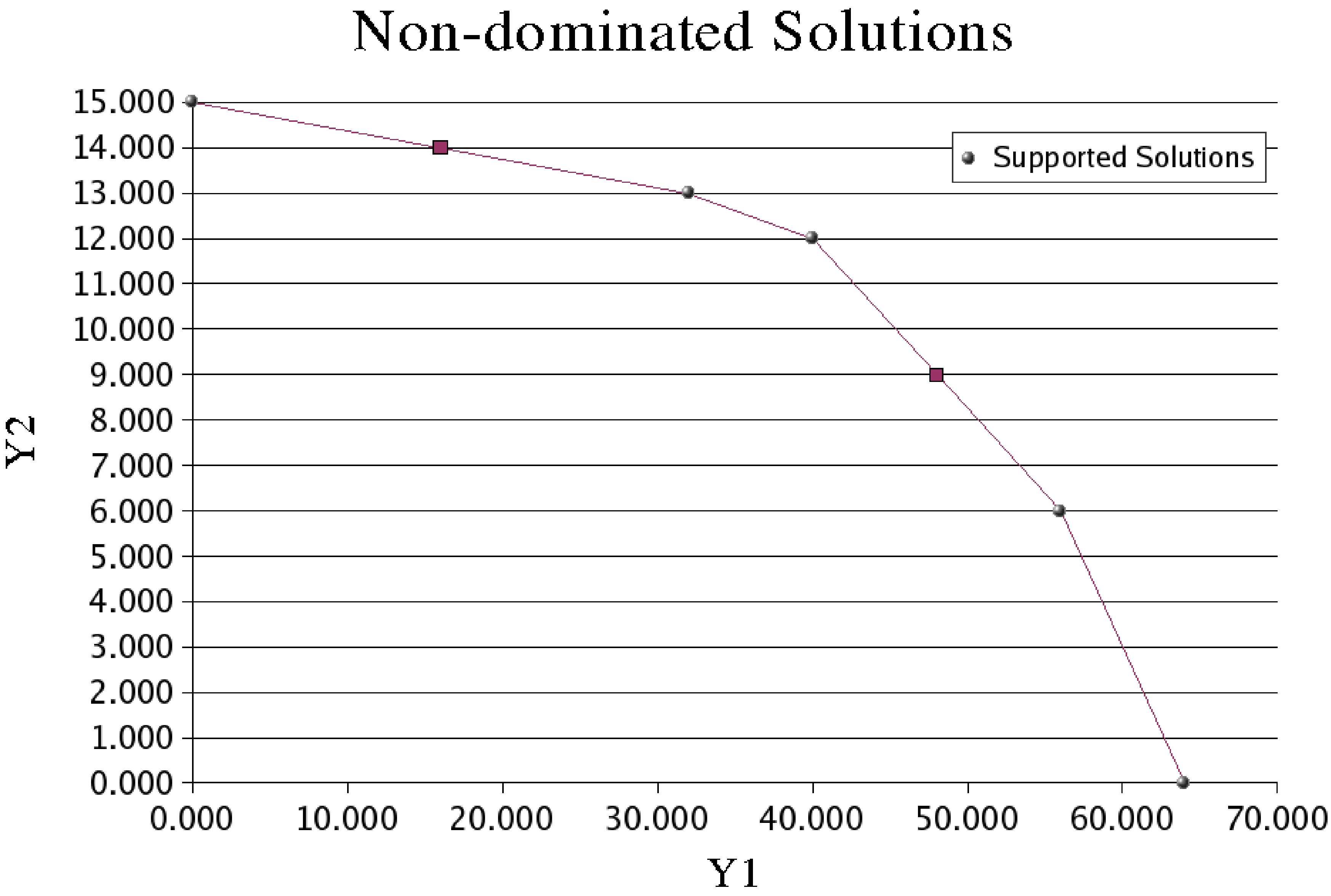
Example: Bicriteria ILP

- Consider the following bicriteria ILP:

$$\begin{aligned} \text{vmax} \quad & [8x_1, x_2] \\ \text{s.t.} \quad & 7x_1 + x_2 \le 56 \\ & 28x_1 + 9x_2 \le 252 \\ & 3x_1 + 7x_2 \le 105 \\ & x_1, x_2 \ge 0 \end{aligned}$$

- For this ILP, we get the set of Pareto outcomes pictured on the next slide.

Example: Pareto and Supported Outcomes for Example

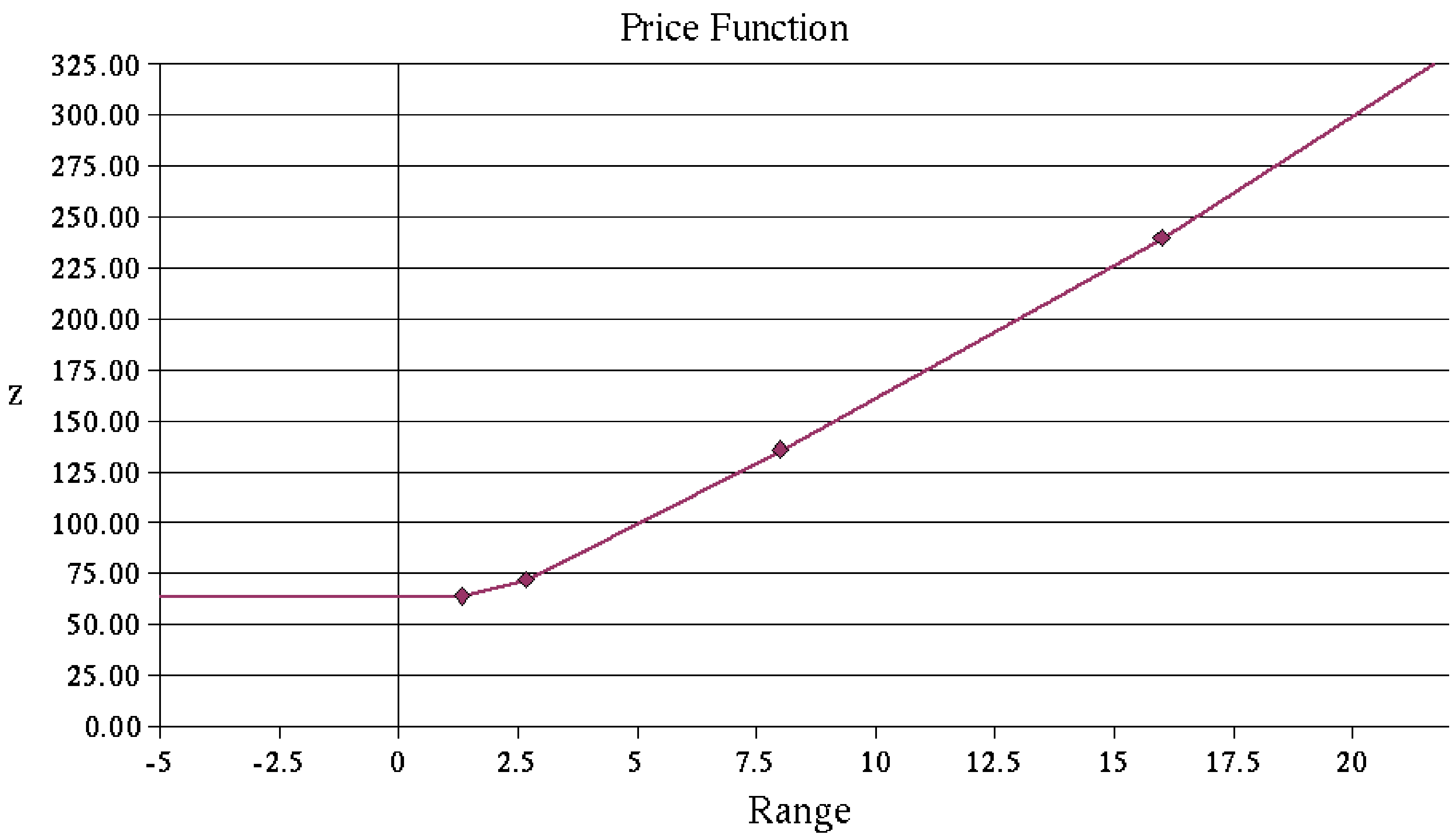


Example: Bicriteria Solver

- Consider the simple MILP from our earlier example.
- By examining the supported solutions and break points, we can easily determine $p(\theta)$, the objective function value as a function of θ.

θ range	$p(\theta)$	x_1^*	x_2^*
$(-\infty, 1.333)$	64	8	0
$(1.333, 2.667)$	$56 + 6\theta$	7	6
$(2.667, 8.000)$	$40 + 12\theta$	5	12
$(8.000, 16.000)$	$32 + 13\theta$	4	13
$(16.000, \infty)$	15θ	0	15

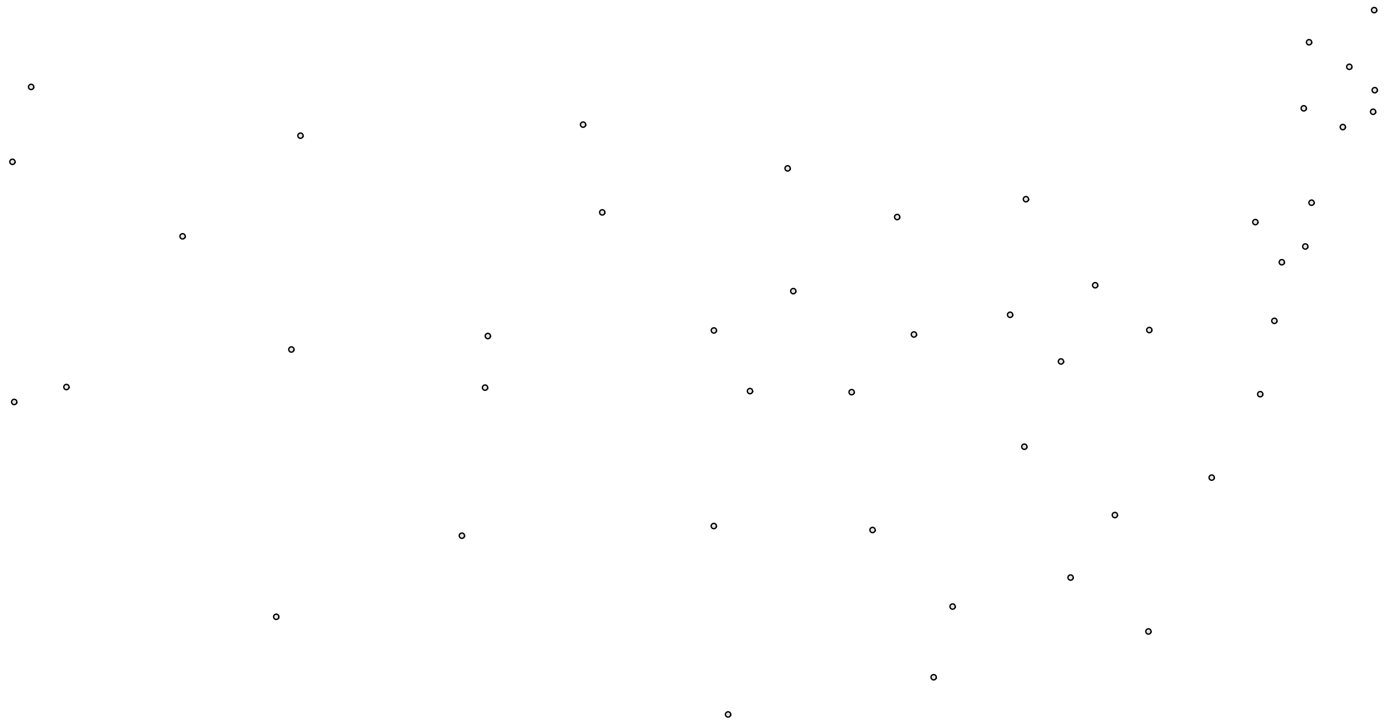
Example: Graph of Price Function



Network Routing Problems

- Using SYMPHONY, we developed a custom solver for a class of network design and routing problems.
- A single commodity is supplied to a set of customers from a single supply point.
- We must design the network and route the demand, obeying capacity and other side constraints.
- We wish to consider both
  - the cost of construction (the sum of lengths of all links), and
  - the latency of the resulting network (the sum of length multiplied by demand carried for all links).
- These are competing objectives, so we can analyze the tradeoff by using the SYMPHONY multicriteria solver.

Example Instance



Conclusion

- We presented a new version of the SYMPHONY solver with an OSI interface supporting warm starting for MILPs.
- We have shown how this capability can be used to implement an efficient bicriteria solver for ILPs.
- We have shown how this solver can in turn be used to perform sensitivity analysis and analyze tradeoffs for competing objectives.
- In future work, we plan on refining SYMPHONY's warm start and sensitivity analysis capabilities.
- Two papers covering the contents of this talk are available.
- Full computational results will be available in a future paper.