Using Nested Column Generation & Generic Programming to solve Staff Scheduling Problems:

Using Compile-time Customisation to create a Flexible C++ Engine for Staff Rostering

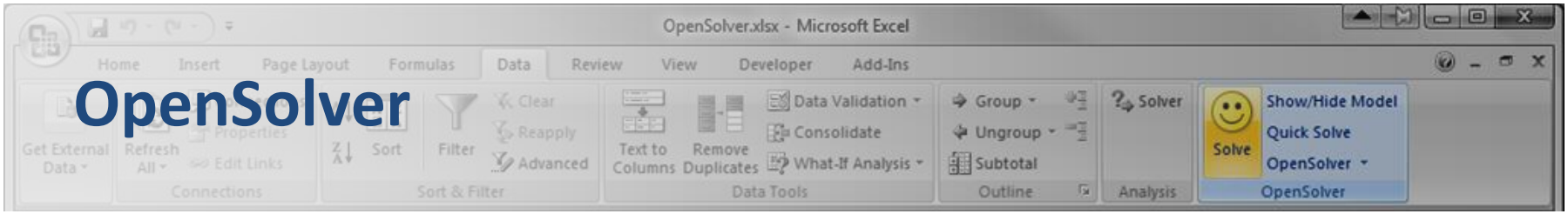
Andrew Mason & Ed Bulog
Department of Engineering Science
University of Auckland
Anders Dohn
Technical University of Denmark
Department of Management Engineering

Integer Programming Down Under: Theory, Algorithms and Applications
July 6—8, 2011
Newcastle NSW Australia

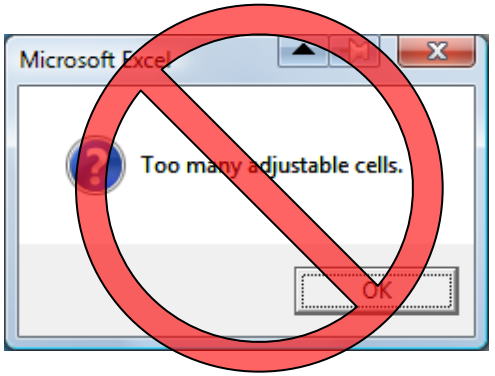
OpenSolver

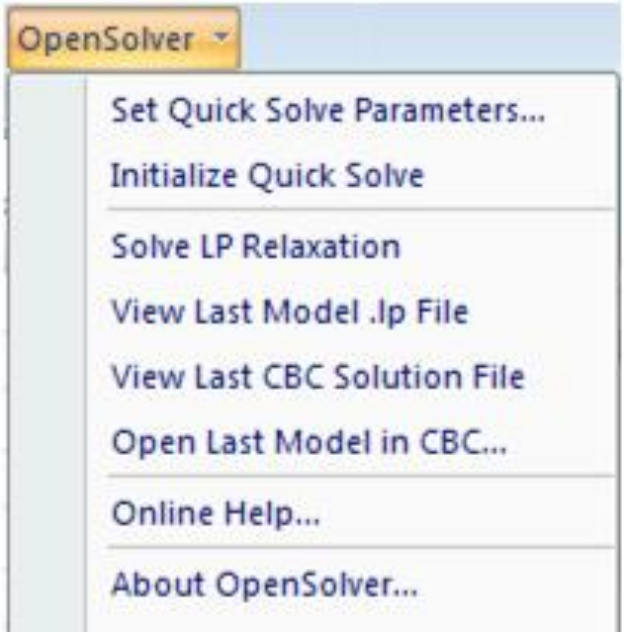
Open source linear programming in Excel using COIN-OR's CBC engine

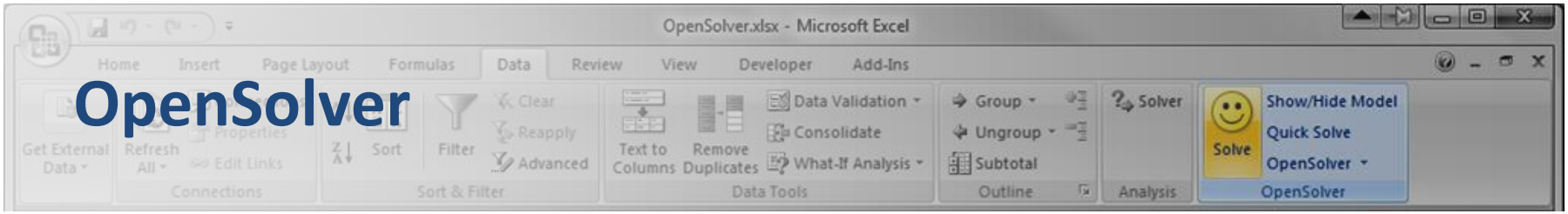
www.opensolver.org



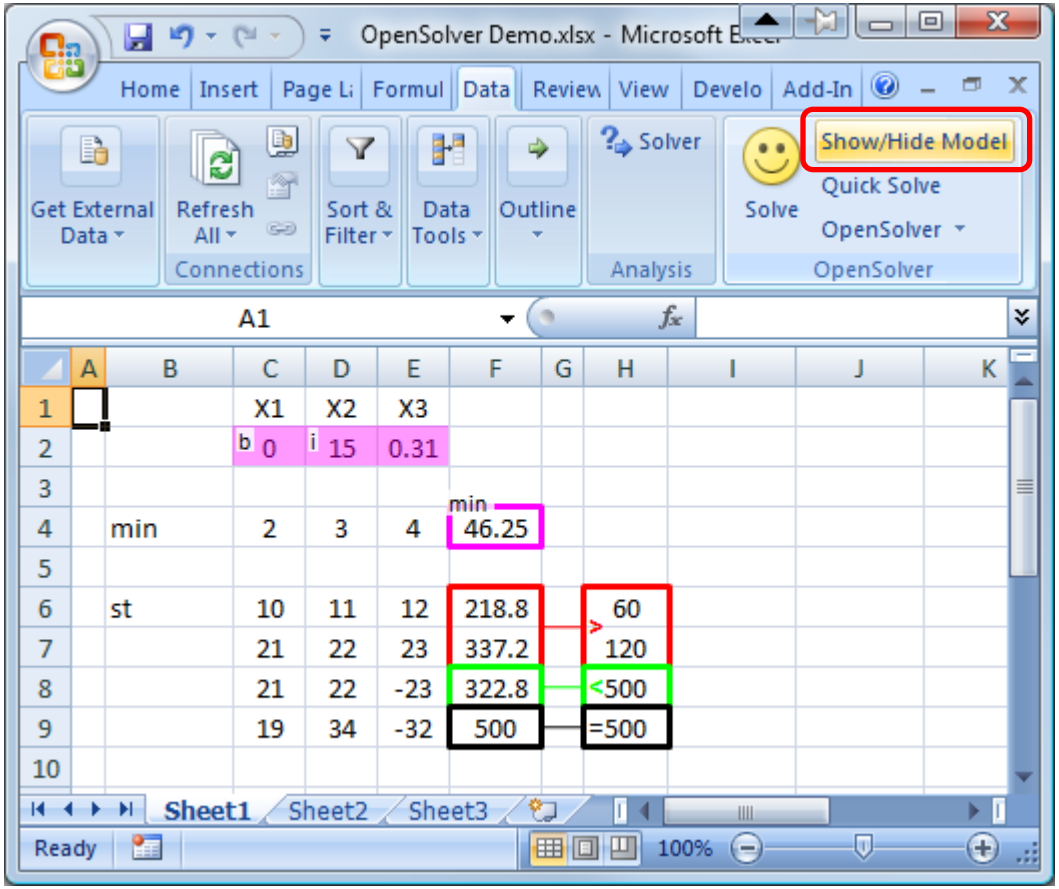
- Free Open Source Excel plugin
- Solver-compatible LP/IP Solver
  - Excel 2003 and later
  - No artificial size limits
- Uses COIN-OR BCP
  - Faster than Solver
- Advanced Features
  - Model visualisation
  - Equation view of model
  - Command line control of CBC
  - …
- AutoModel and better GUI
  - Next release

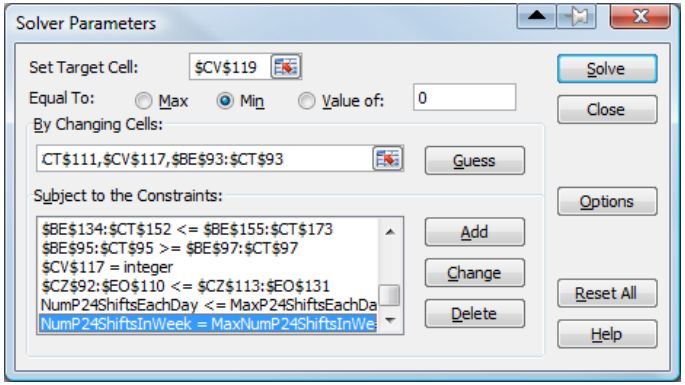






On-sheet Model Visualisation





Available at:
opensolver.org

Using Nested Column Generation & Generic Programming to solve Staff Scheduling Problems:

Using Compile-time Customisation to create a Flexible C++ Engine for Staff Rostering

Andrew Mason & Ed Bulog
Department of Engineering Science
University of Auckland
Anders Dohn
Technical University of Denmark
Department of Management Engineering

Integer Programming Down Under: Theory, Algorithms and Applications
July 6—8, 2011
Newcastle NSW Australia

Optimised Dynamic Rosters

- Rosters are constructed ‘just in time’ to match a particular period’s requirements

	m	t	w	t	f	s	s	m	t	w	t	f	s	s	m	t	w	t	f	s	s	m	t	w	t	f	s	s
Peter	D	D	D	D	D					N	N	N	N	N	?													
Paul	D	D	D	D			N	N	N	N	N	N																
Jo				N	N			N	N				D	D														
Mike	N	N	A	A	A	A				D	D	D	D	D														
Sue	N	N	N				A	A	A	D	D	D																
Tom				D	D	D	D	D	D				N	N														

- Applications for Dynamic Optimised Rosters:
  - Nurse Rostering
  - Casinos
  - Call Centres
  - Airlines etc

Example – Nurse Rostering

- 28 day roster period
- Five different shifts a day:
  - M, A, N, 6, 8
  - All shifts 8 hours in duration
- Contract specifies paid hours requirements:
  - 80, 72, 64, 56, 40, 32, 30, or 28 hours/fortnight
- Management specifies work requirements:

Week Days					Weekends				
M	A	N	6	8	M	A	N	6	8
≥ 4	≥ 10	=5	≥ 9	=13	=4	=8	=3	≥ 6	=8

Example – Nurse Rostering

- Complex rules and quality measures
  - Max number of days on in succession or in a week.
  - Some combinations of on/off days prohibited.
  - A minimum rest period after a shift is required.
  - Specific shift transitions are not allowed.
  - Split weekends are undesirable.
  - Single days-on / days-off are undesirable.
- Staff members can have individual preferences for shifts, days-on, days off
- Ignore: staff skills, shifts of different lengths, and work requirements relating to overlapped shifts

Example Solution – 28 day Roster

	1 M	2 T	3 W	4 Th	5 F	6 Sa	7 Su	8 M	9 T	10 W	11 Th	12 F	13 Sa	14 Su	Hrs
Nurse 1	**M**	**M**	**M**	**M**			**N**	**N**	**N**	**N**	**N**				**72**
Nurse 2		N	N	N	N	N	N				M	M	M	M	80
Nurse 3	6	6	6	6			8	8	8	8			N	N	56
…															
Nurse 86	A	A	A	A						D	D	D	D		64

	15 M	16 T	17 W	18 Th	19 F	20 Sa	21 Su	22 M	23 T	24 W	25 Th	26 F	27 Sa	28 Su	Hrs
Nurse 1	**6**	**6**	**6**	**6**	**6**	**6**		**6**	**6**		**A**	**A**	**A**	**A**	**72**
Nurse 2	M			M	M	M	M	M	M			A	A	A	80
Nurse 3	N							N	N	N	N	N	N		56
…															
Nurse 86			N	N	N	N				A	A	A	A		64

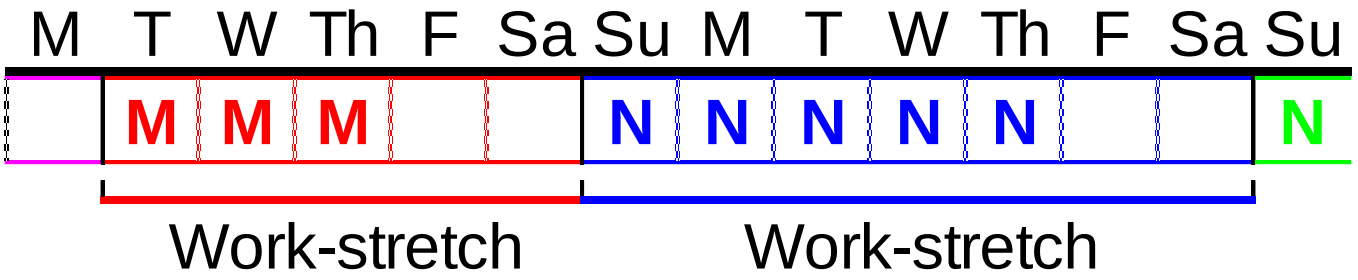
Each nurse works their own personalised 28-day *roster-line*

Set Partitioning Formulation

		Nurse 1				Nurse 2					Nurse n					
	Nurse 1	1	1	…	1										=	1
	Nurse 2					1	1	…	1						=	1
	⋮									⋮					⋮	⋮
	Nurse n										1	1	…	1	=	1
Mon	M	1				1	1							1	≥	4
	A										1				≥	10
	N														=	5
	8		1									1			≥	9
	6				1				1			1			=	13
Tue	M													1	≥	4
	A					1					1				≥	10
	N	1					1								=	5
	8		1									1			≥	9
	6				1				1			1			=	13
	⋮	⋮		⋮		⋮		⋮	⋮		⋮		⋮	⋮	⋮	⋮
Sun	M													1	≥	4
	A					1					1				≥	10
	N	1					1								=	5
	8		1									1			≥	9
	6				1				1			1			=	13

Nested Column Generation Approach

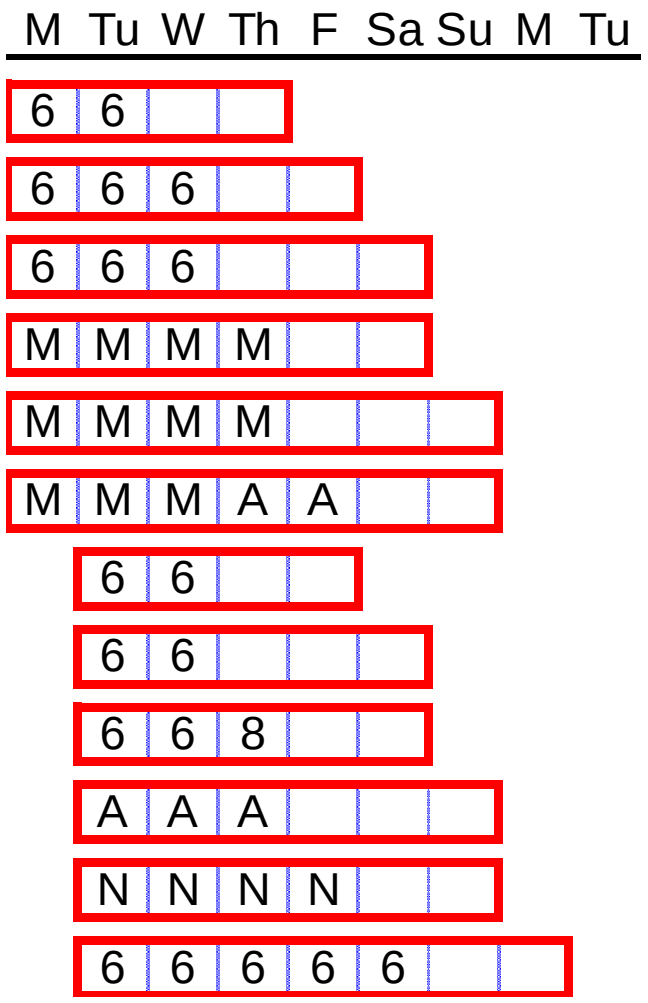
- An individual staff member's roster line is composed of *work-stretches*...



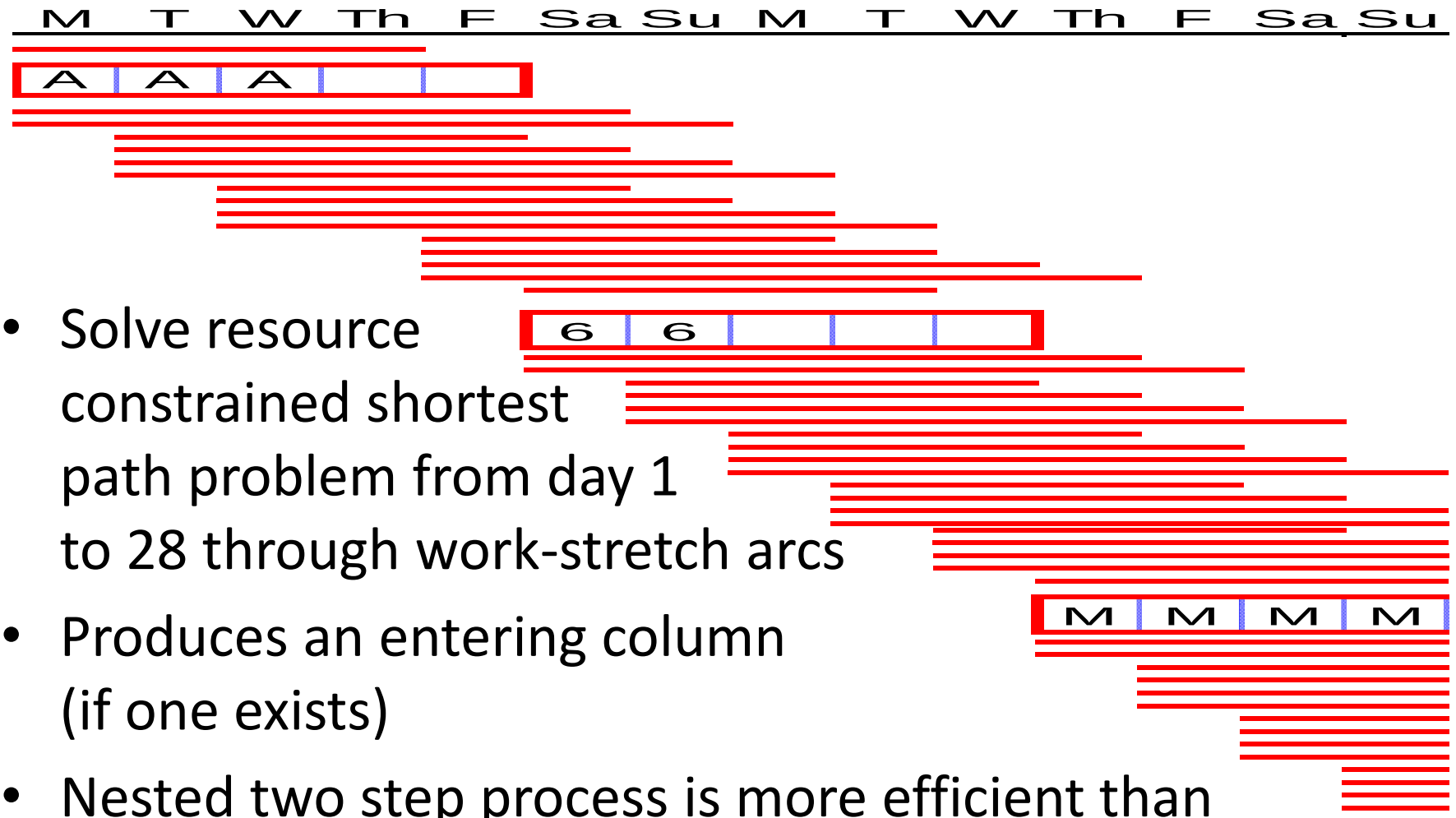
- Roster-line quality ≈ sum of work-stretch qualities
- Use a nested column generation approach:
  - Problem 1: Generate many good work-stretches
  - Problem 2: Combine work-stretches to generate a roster-line (i.e. an entering column)

Generating Columns: Step 1- Build Work-stretches

- Find most negative reduced cost workstretches for staff member *i*
- One best workstretch for each combination of:
  - start day
  - end day
  - hours worked
  - (any other resources)
- Need to solve resource constrained shortest path problems
  - fast to solve as nested



Step 2 - Combine work-stretches to form the best roster-line

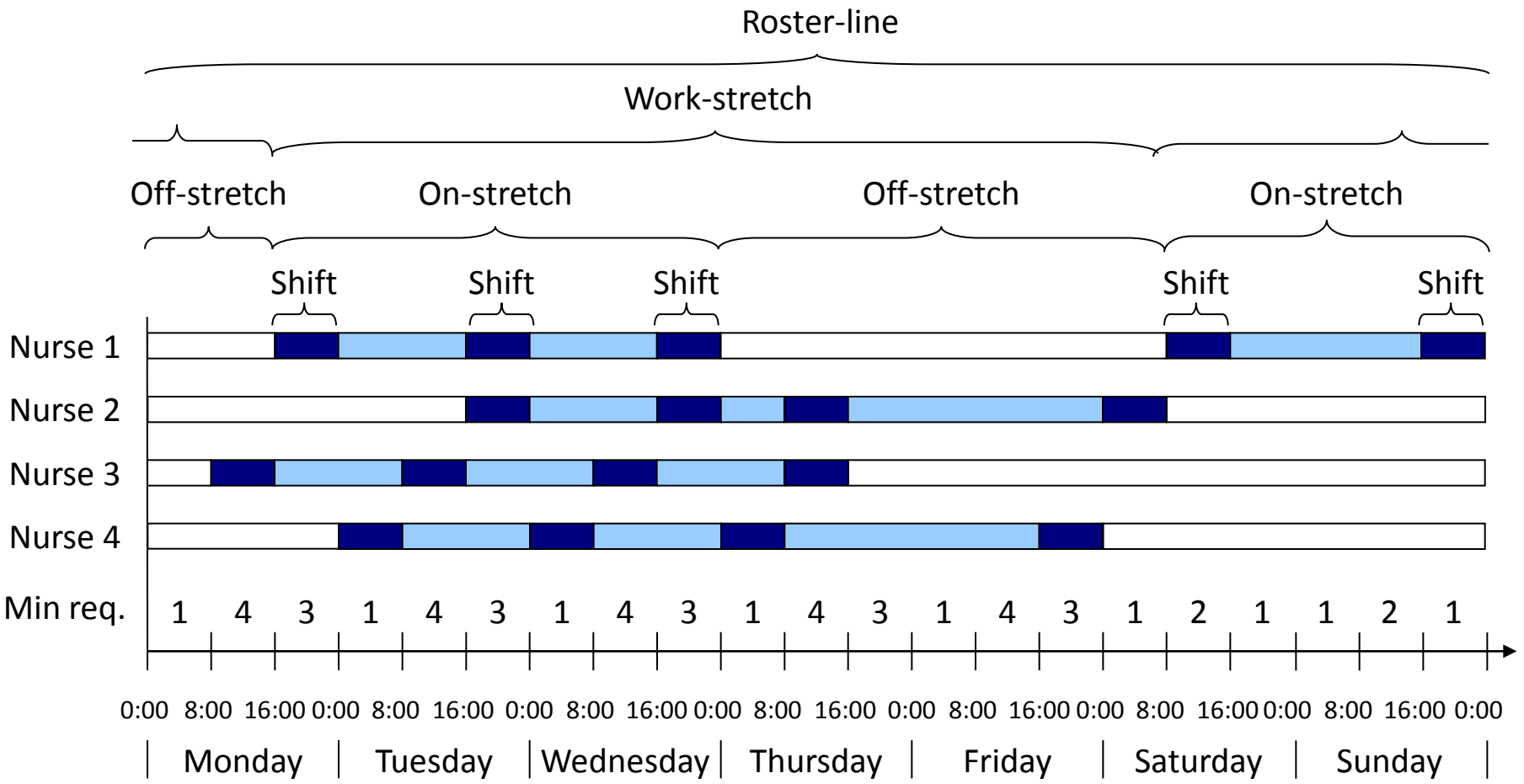


- Solve resource constrained shortest path problem from day 1 to 28 through work-stretch arcs
- Produces an entering column (if one exists)
- Nested two step process is more efficient than solving a single large problem

Progressing towards Generality

- Engineering Science Dept contributions…
  - Mark Smith – Nurse Rostering
    - Introduced nested column generation → fast run times
    - Hard to generalise for a new problem
  - David Nielsen – TabCorp rostering (Melbourne Cup)
    - Introduced an internal rule modeling language – easier to generalise
    - Did not use column generation, so suited to small problems only.
  - Faram G. Engineer – Flexible C++ Rostering Engine
    - Combines nested column generation with rule modelling flexibility
    - Proven ability to solve a wide range of problems
    - Implements a more nested column generator…

Nested Column Generation



Nested Column Structure

- Consider a roster line:

	M	T	W	T	F	S	S	M	T	W	T	F	S	S	M	T	W	T	F	S	S	M	T	W	T	F	S	S
Nurse 1	M	M	-	N	-	M	M	M	-	-	N	N	-	-	M	M	M	M	M	M	-	-	-	-	M	M	M	N

- Constructing this line might include the steps:

Nested Column Structure

- Consider a roster line:

	M	T	W	T	F	S	S	M	T	W	T	F	S	S	M	T	W	T	F	S	S	M	T	W	T	F	S	S
Nurse 1	M	M	-	N	-	M	M	M	-	-	N	N	-	-	M	M	M	M	M	M	-	-	-	-	M	M	M	N

- Constructing this line might include the steps:

OnStretch + Shift → OnStretch:

M M + *M* → M M *M*

Nested Column Structure

- Consider a roster line:

	M	T	W	T	F	S	S	M	T	W	T	F	S	S	M	T	W	T	F	S	S	M	T	W	T	F	S	S
Nurse 1	M	M	-	N	-	M	M	M	-	-	N	N	-	-	M	M	M	M	M	M	-	-	-	-	M	M	M	N

- Constructing this line might include the steps:

OnStretch + Shift → OnStretch:

M	M	+	*M*	→	M	M	*M*

OnStretch + OffStretch → Work Stretch:

M	M	M	+	-	-	→	M	M	M	-	-

Nested Column Structure

- Consider a roster line:

	M	T	W	T	F	S	S	M	T	W	T	F	S	S	M	T	W	T	F	S	S	M	T	W	T	F	S	S
Nurse 1	M	M	-	N	-	M	M	M	-	-	N	N	-	-	M	M	M	M	M	M	-	-	-	-	M	M	M	N

- Constructing this line might include the steps:

OnStretch + Shift → OnStretch:

M M + *M* → M M *M*

OnStretch + OffStretch → Work Stretch:

M M M + - - → M M M - -

Roster line + Work Stretch → (Longer) Roster line:

M M - N - + M M M - - → M M - N - *M M M* - -

Nested Column Structure

- Could construct lists of all possible entities, hence giving the best roster-line (=entering column)

OnStretch + Shift → OnStretch:

M M + *M* → M M *M*

OnStretch + OffStretch → Work Stretch:

M M M + - - → M M M - -

Roster line + Work Stretch → (Longer) Roster line:

M M - N - + M M M - - → M M - N - *M M M* - -

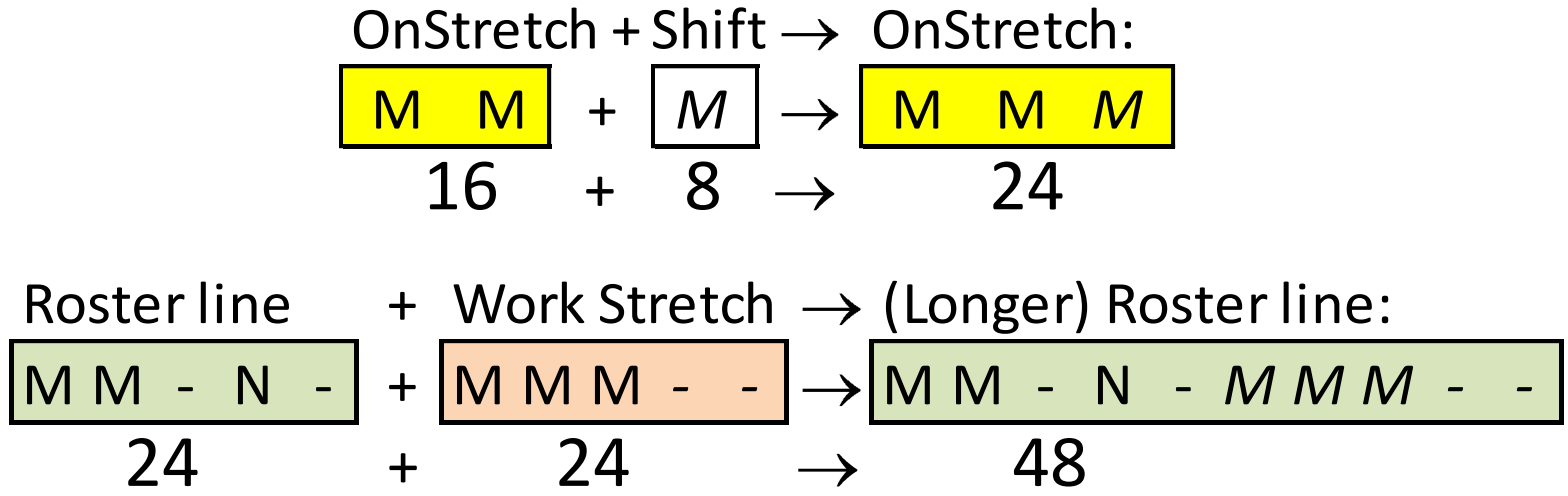
- But, if we store these cleverly – introducing states & dominance rules, and keeping only the best - we have a nested dynamic programming framework for solving resource constrained shortest paths.

Nested Column Structure

- Legality rules and roster quality measures can be expressed in terms of these entities and associated *attribute (resource) values*
- Example:
  - ‘The paid hours must be 72 in each fortnight’
  - To enforce this, we have to track the paid hours within each fortnight

Attribute Value Calculations

- Example: 'Paid Hours' attribute:



  - More complex calculations needed for other attributes
- Attributes allow us to determine:
  - Determine legality and quality (ie cost) of entities, and
  - Implement dominance rules

Software Implementations

Faram Engineer's C++ Code:

- Shifts, on-stretch, off-stretch, work-stretch, roster-line classes
  - Well defined rules for creating sets of these
  - User defined attributes, with rules for calculating values & dominance
- Powerful, but lots of C++ run-time overhead

Anders Dohn's Generic Programming C++ Code:

- Compile-time Customisation = Generic Pro… STL)
- Customised C++ code prod…
- Customisati… extensions
  …g language
- …defines a roster problem
- …stomised code fully optimised at compile time - fast

Generic Programming Column Generator:
20 times faster

New Customised Code Framework

User wants a shift to have *start-time*, *end-time*, & *paid-hours*:

```
# define SHIFT_ATTRIBUTES \
    ATT( (starttime   , int, "Starttime"), \
    ATT( (endtime     , int, "Endtime")  , \
    ATT( (paidhours , int, "PaidHours"), \
    END )))
```

This is compiled to give C++ code with these attributes:

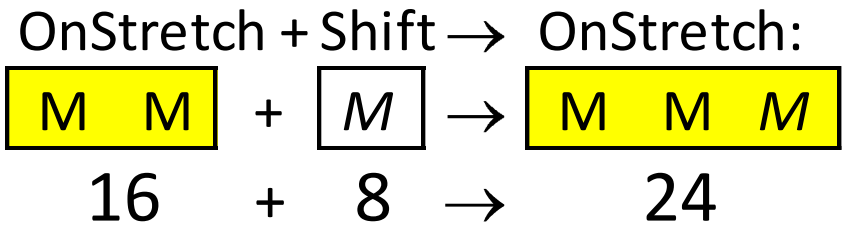
```
class Shift {
public:
    Attribute<10, int, ... > starttime;
    Attribute<11, int, ... > endtime;
    Attribute<12, int, ... > paidhours;
}
```

New Customised Code Framework

User wants an OnStretch to have a *paidhours*...

```
ATT( (paidhours, int, "paidhours", ...,
    o.paidhours + s.paidhours, ....) \
```

.. to be calculated by adding *paidhours:*



The user's calculation defined above is inserted into the code:

```
OnStretch() {
    paidhours = o.paidhours + s.paidhours;
}
```

New Customised Code Framework

Value of *paidhours* can control feasibility, dominance & cost:

```
ATT( (paidhours, int, "paidhours", feas_all, domi_exact, cost_none,
    o.paidhours + s.paidhours, s.paidhours) \
```

The highlighted terms determine:

– *feas_all*: What values are feasible for this attribute

– *domi_exact*: Controls dominance within the column generator

– *cost_none*: How this attribute contributes to the cost (quality) of the on-stretch

Results

- Implemented in COIN-OR BCP framework
- Tested on 3 real problems
- Successfully modelled all rostering rules
- Restricted generation runs => heuristic

Problem	NZ	Denmark1	Denmark2
Number of Staff	85	28	40
Number of Shift Types	5	4	18
Scheduling Period	4 weeks	4 weeks	4 weeks
Heuristic root node LP value	19.667	288.822	1.5
Heuristic first integer solution value	23	296	12
Heuristic best feasible integer value	23	281	1
Seconds in root node	7.5	89.63	73.48
Seconds to find 1st integer solution	23.95	180.77	509.92
Seconds to find best integer solution	23.95	186.2	532.64
Total runtime	26.91	192.56	552.16
Runtime Split: Solving LP	41.30%	10.20%	79.10%
Branching	1.20%	0.20%	0.20%
Overhead	13.10%	3.80%	4.30%
Pricing	44.50%	85.80%	16.40%
Pricing Time Split: Setup	5.00%	1.40%	0.80%
On-stretch	6.10%	0.70%	5.90%
Work-stretch	25.70%	1.20%	8.60%
Rosterline	7.70%	82.60%	1.10%
Tree size	771	371	675
Max depth	383	185	280
Pricing problems solved	4,445	1,489	4,800
Columns generated	1,919	7,982	19,576
True root LP value	19.667	234	1
Optimal IP solution value	23	(235)	1
Seconds to find true root LP value	8.08	2,024.72	2,639.36
Seconds to find optimal IP solution	27.48	> 10 h	5,652.58

First International Nurse Rostering Competition 2010

- See www.kuleuven-kortrijk.be/nrpcompetition
  - CODeS research group, Katholieke Universiteit Leuven, Belgium
  - SINTEF research group in Norway
  - University of Udine, Italy.
  - Results presented at PATAT 2010.
- Random rostering problem instances:
  - Sprint Instances (about 10 seconds allowed)
  - Medium Distance Instances (about 10 minutes allowed)
  - Long Distance Instances (about 10 hours allowed)
- Only two hard constraints to satisfy:
  - All shift demands must be met at equality
  - Each nurse can work at most one shift (starting) on each day
- Other ‘soft’ constraints – violations penalised in objective

First International Nurse Rostering Competition 2010

- The objective is to minimise the weighted sum of all soft-constraint violations, which may include:
  - Min/Max number of
    - total shift assignments
    - consecutive shift assignments and consecutive days off
    - consecutive working weekends
  - Max working weekends in four weeks
  - Complete working weekends
  - Identical shift types during a weekend
  - Exclusion of unwanted patterns of shifts
    - For example, working a Night shift before a free weekend
- Successfully modelled all of these using our system

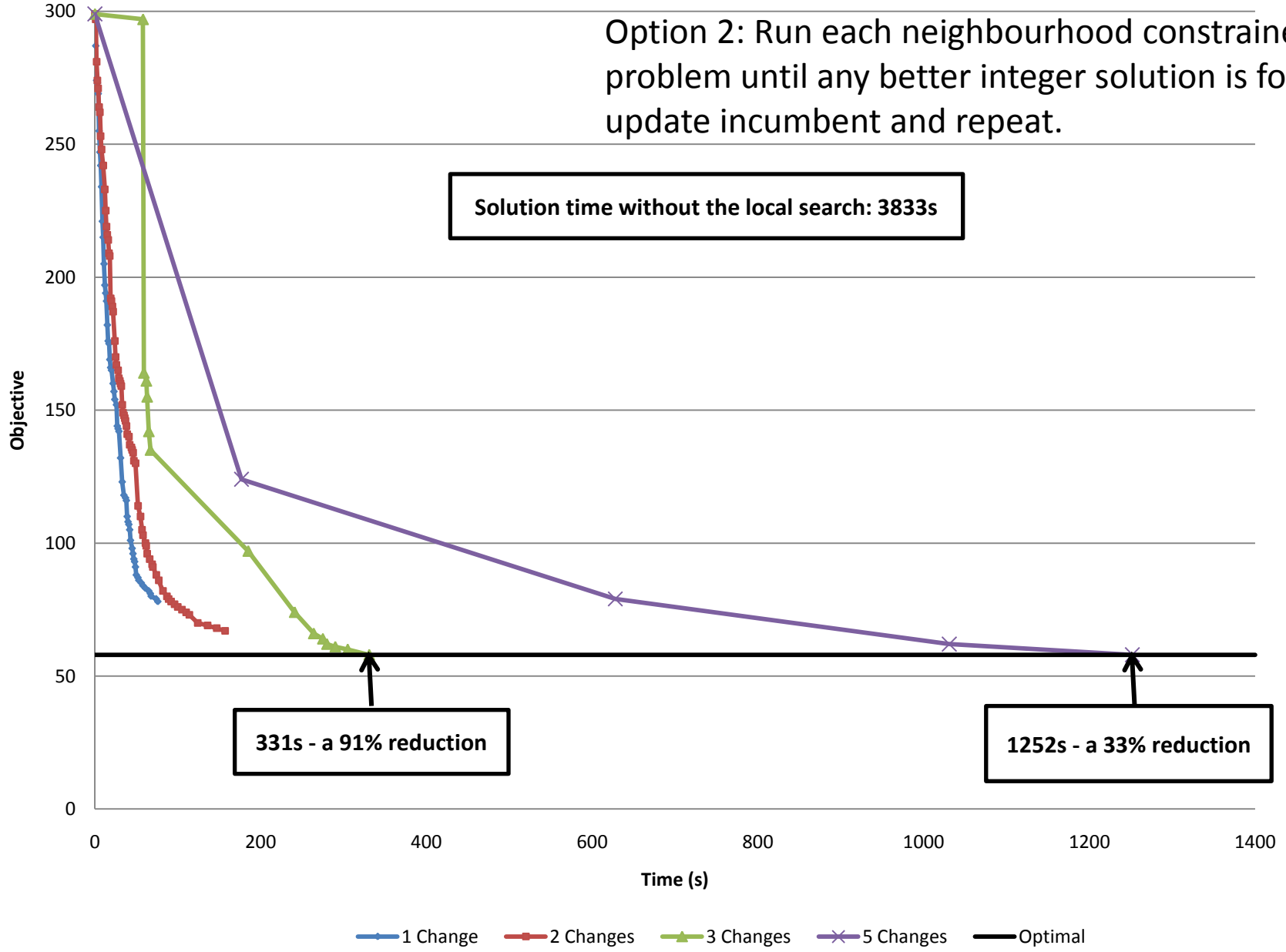
Reducing Run Times

- Local Search using Neighbourhood Column Generation
  - Apply neighbourhood restrictions within generator
  - Perhaps better for problems with large IP gaps?
- Elastic Constraint Branches
  - Prevent fractionation across multiple optimal solutions
- Dual Stabilisation
  - Add this to the COIN-OR BCP software (work in progress)
  - What is the state of the art for the update regime?

Neighbourhood Column Generation

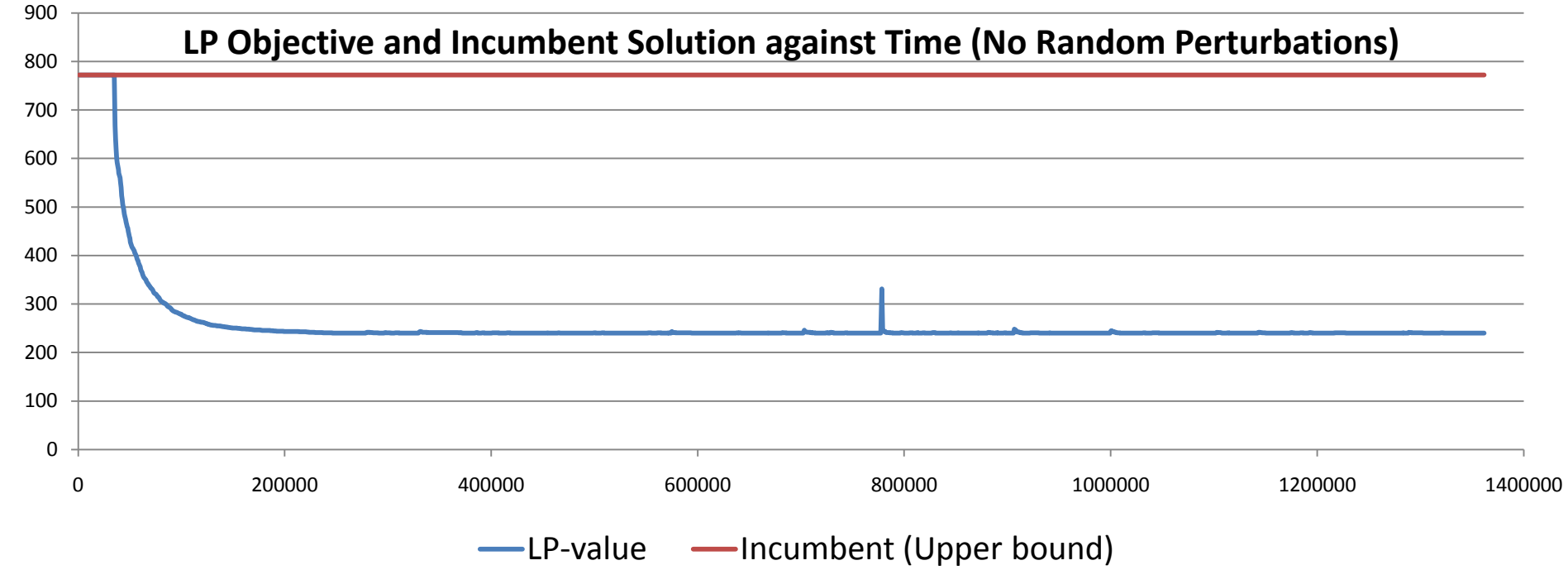
- Each nurse is assigned an incumbent roster-line from some initial feasible solution
- Neighbourhood column generation
  - For each person, generate roster-lines which differ by at most *N* shifts from their current incumbent roster-line
- Solve to find *any* better integer solution and then update incumbent
- Repeat until no improvement found

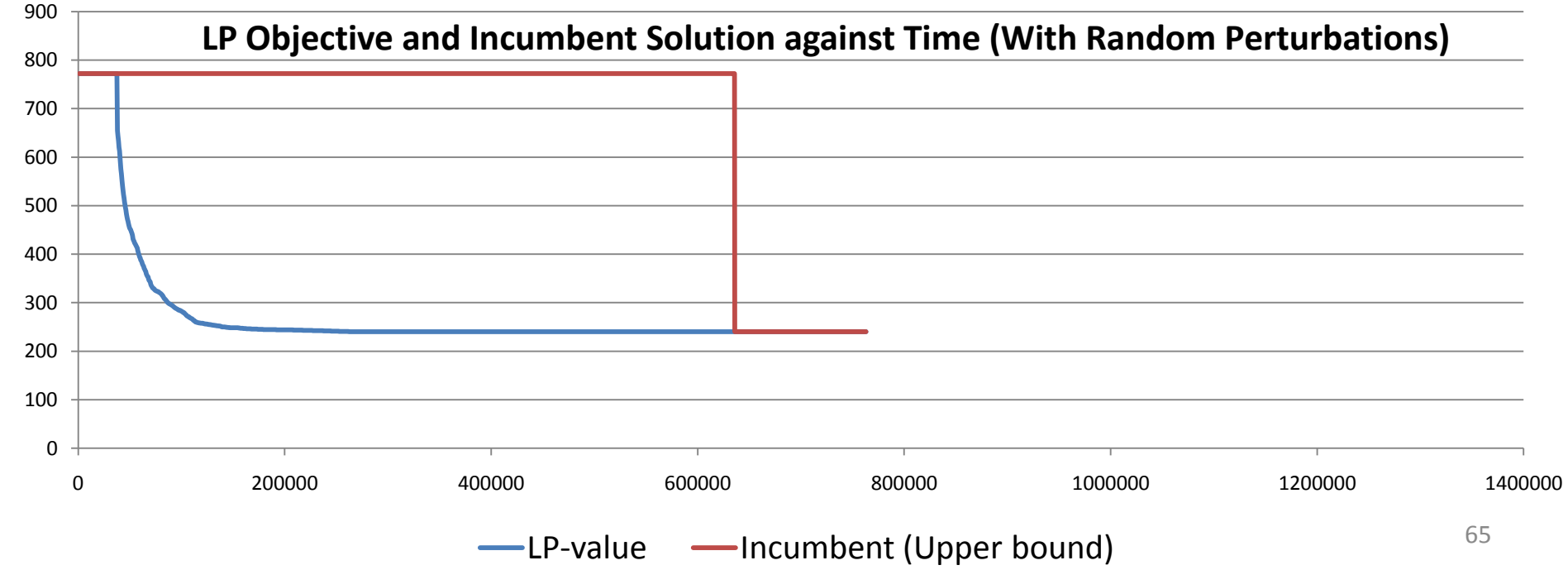
Neighbourhood Column Generation: Objective Function Values against Time



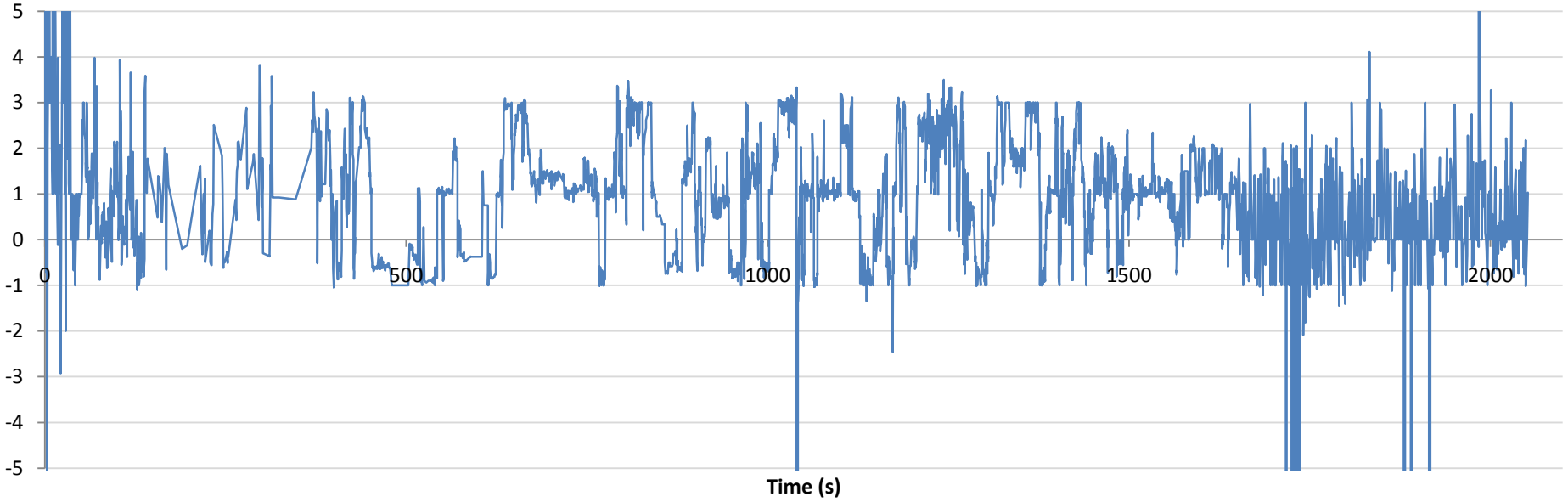
Elastic Constraint Branches

- There are many equivalent columns for each employee
  - Caused by equivalent shift choices facing each employee
- Introduce small random costs (preferences) for each employee-shift assignment
  - Allows generator to differentiate between shifts
  - Gives different shifts to different staff
  - Small cost perturbations do not affect the optimal solution, but guide solution towards a particular optimal solution
- These are ‘elastic constraint branches’ as implemented when extending Wedelin Algorithm to rostering problems

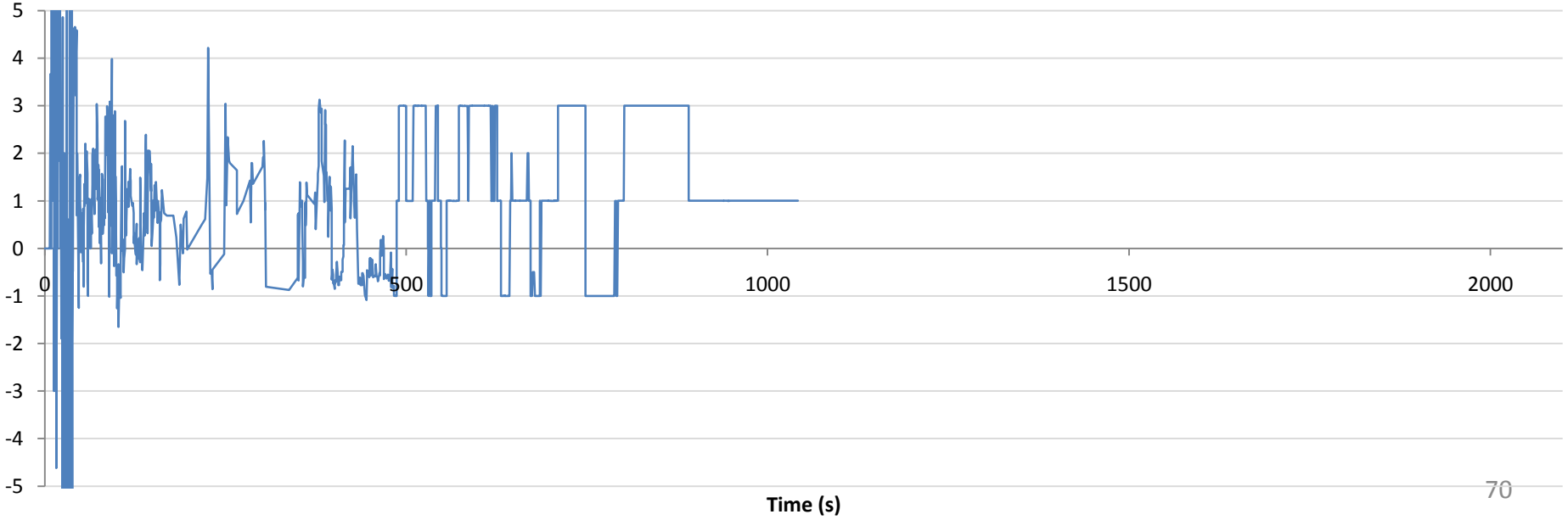

LP Objective and Incumbent Solution against Time (No Random Perturbations)
900
800
700
600
500
400
300
200
100
0
0
200000
400000
600000
800000
1000000
1200000
1400000
LP-value
Incumbent (Upper bound)


LP Objective and Incumbent Solution against Time (With Random Perturbations)
900
800
700
600
500
400
300
200
100
0
0
200000
400000
600000
800000
1000000
1200000
1400000
LP-value
Incumbent (Upper bound)

Dual for Shift Constraint #10 (No Random Perturbations)



Dual for Shift Constraint #10 (With Random Perturbations)



Competition Problem Results

Instance Name	Our Standard Approach	Random Cost Perturbations	Neighbourhood Column Generation		Neighbourhood Col Gen and Random Cost Perturbation	
			3 change neighbourhood	5 change neighbourhood	3 change neighbourhood	5 change neighbourhood
sprint01	56* (1709s)	56* (618s)	57 (192s)	57 (149s)	56* (579s)	56* (588s)
sprint02	58* (2087s)	58* (1339s)	62 (191s)	58* (609s)	59 (473s)	59 (495s)
sprint03	51* (1555s)	51* (691s)	55 (188s)	51* (2065s)	53 (1121s)	51* (1777s)
medium01	240* (1265s)	240* (541s)	243 (754s)	242 (1316s)	246 (3222s)	240* (1193s)
medium02	240* (1372s)	240* (456s)	244 (451s)	240* (684s)	243 (4509s)	240* (728s)
medium03	236* (2619s)	236* (976s)	242 (599s)	237 (528s)	238 (5264s)	236* (1745s)
long01	out of memory	197* (777s)	206 (832s)	197* (1956s)	201 (5402s)	197* (1356s)
long02	out of memory	219* (827s)	230 (678s)	221 (5580s)	228 (7576s)	220 (2447s)
long03	out of memory	240* (760s)	241 (1039s)	240* (1147s)	240* (7303s)	240* (1483s)

Instance Type	Number of Staff	Number of Rows	Number of Days	Number of Shift Types	Number of Shifts	Number of Contracts	Number of Patterns	Number of Requests
sprint	10	162	28	4	152	4	3	450
medium	31	639	28	4	608	4	0	682
long	49	749	28	5	700	3	3	2695

Legend

Objective *=optimal
(Time to find best integer solution)

Best performer

Conclusions

- Flexible rostering software developed
  - Powerful modelling framework
    - Complex attribute handling to model roster rules
    - Can model all problems tested
  - Efficient nested column generator
    - Exploits problem structure, but still easy to customise
  - Framework implemented using generic programming
    - Compile-time customisation => 20x faster
  - Neighbourhood column generation
    - Faster solve times for some problems
  - Elastic constraint branches help integrality