MONASH
University

Solver-independent Large Neighbourhoud Search

Jip J. Dekker, Maria Garcia de la Banda, Andreas Schutt, Peter J. Stuckey, and Guido Tack

Large Neighbourhood Search

- *Meta-search* (Shaw, 1998)
- Given a solution, uses a meta-heuristic to iteratively:
  - Relax part of the solution (the neighbourhood)
  - Search for another solution in that neighbourhood
- State of the art in, e.g.:
  - Vehicle Scheduling Problems
  - Timetabling Problems

Complete Methods with LNS

- Can be combined with complete methods
- This can **quickly improve** found solutions and provides good **scalability**

Complete Methods with LNS

Algorithm 1 Large Neighbourhood Search

1: **procedure** LNS($P = (C, X, D, f)$)
2: $a \leftarrow findsolution(P)$
3: **while** $\neg timeout()$ **do**
4: $P' = (C \cup nbh(a) \cup obj(a), X, D, f)$
5: $a' \leftarrow findsolution(P')$
6: **if** a' is a solution **then** $a \leftarrow a'$
7: **return** a

Other LNS approaches

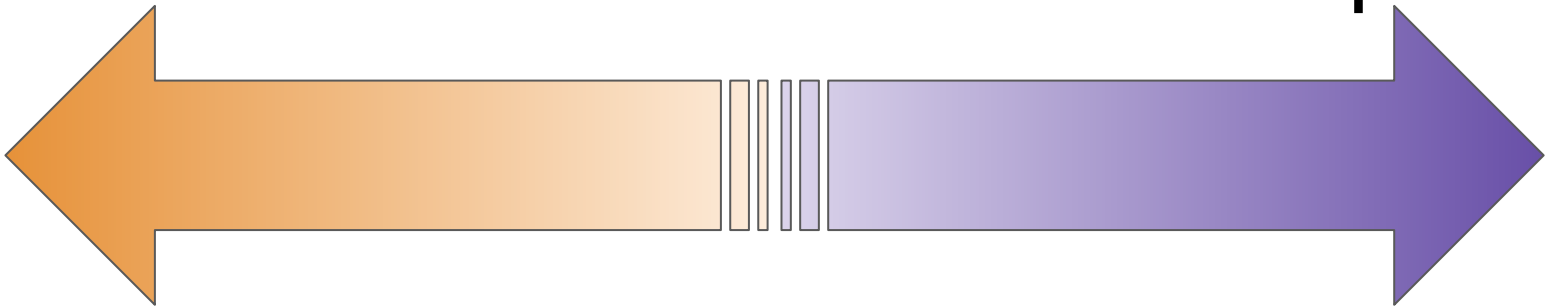
Other meta-heuristics can be used in LNS:

- Multiple Neighbourhoods
  - Round Robin
  - Adaptive Selection
- Simulated Annealing
- Tabu Search

Modelling with LNS

Solver internal LNS

Scripted LNS



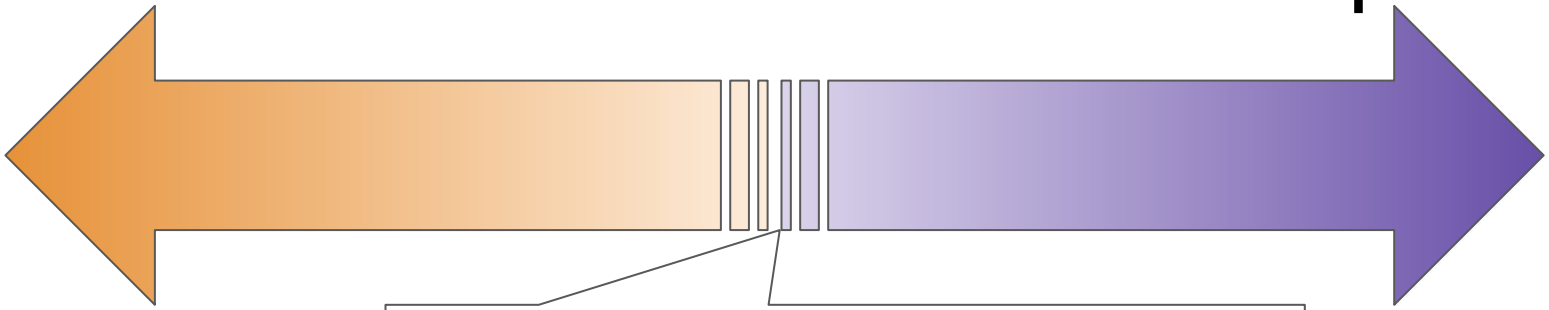
- Ideal performance
- Neighbourhoods internal to the solver

- Flexible neighbourhoods
- Requires repeated runs of the solver

Modelling with LNS

Solver internal LNS

Scripted LNS



Compromise:

Modelling driven LNS

MiniSearch

- MiniZinc’s initial approach to **Meta-Search**
- LNS in MiniSearch consists of two parts:
  - Meta-heuristic definition (MiniSearch Function)
  - Neighbourhood definition (MiniZinc Predicate)

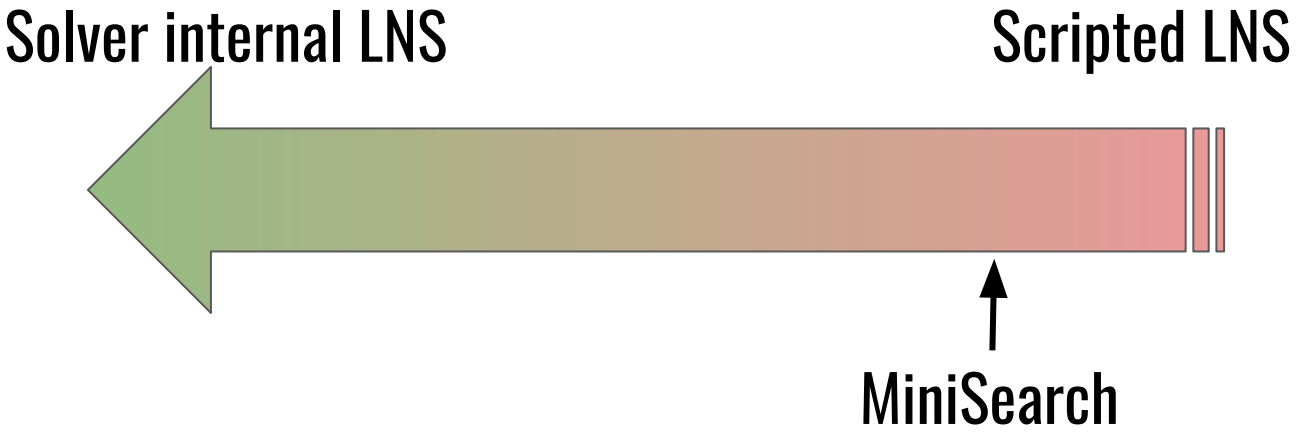
LNS in MiniSearch

```
function ann: lns(var int: obj, array[int] of var int: vars,
                  int: iterations, float: destrRate, int: exploreTime) =
  repeat (i in 1..iterations) ( scope(
      if has_sol() then post(uniformNeighbourhood(vars,destrRate))
      else true endif /\
      time_limit(exploreTime, minimize_bab(obj)) /\
      commit() /\ print()
    ) /\ post(obj < sol(obj)) );
```

Random Neighbourhood

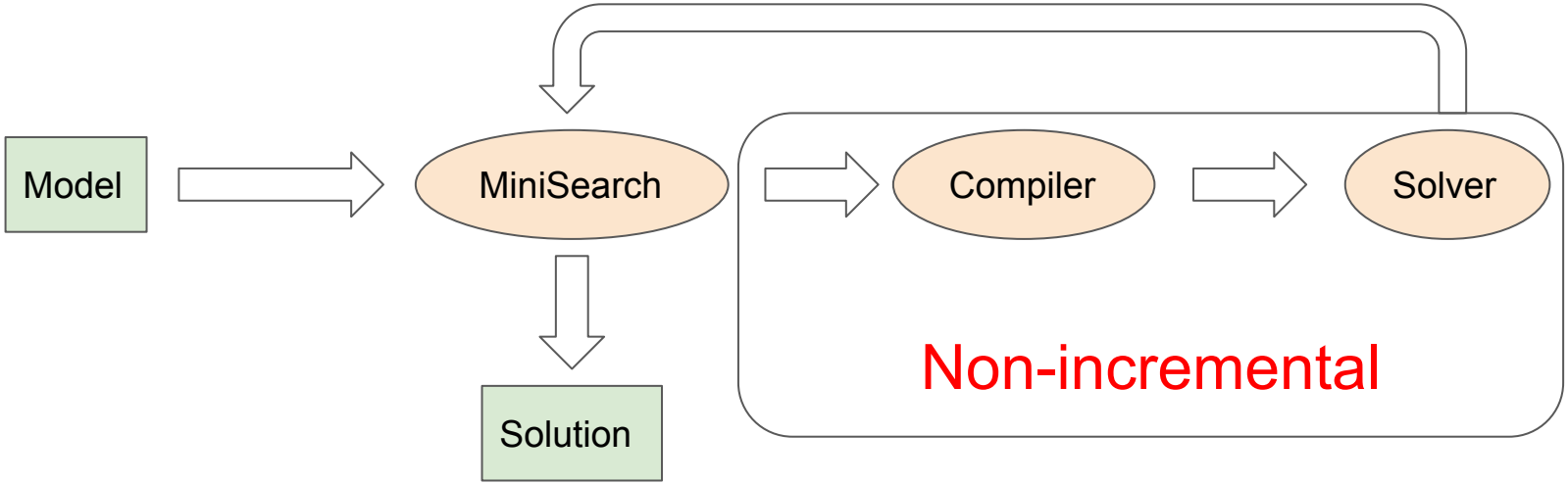
```
predicate uniform_nbh
(array[int] of var int: x,float: destrRate)

= forall(i in index_set(x))(
    uniform(0.0,1.0) > destrRate
      -> x[i] = sol(x[i])
);
```

MiniSearch Drawbacks

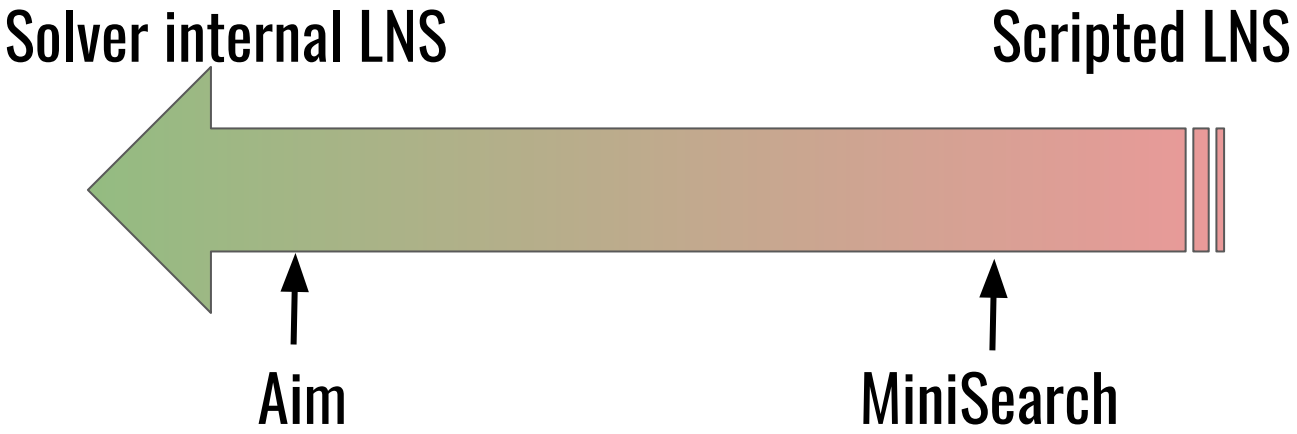


- Performance similar to Scripted LNS
- Search definitions can be *unintuitive* and *unwieldy*

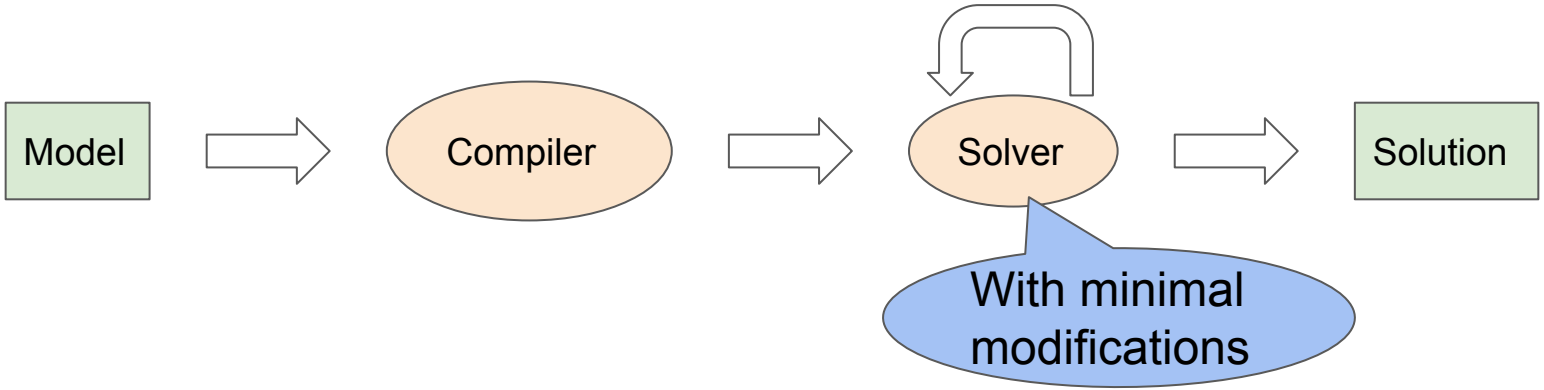
MiniSearch Approach



Our Aim



A New Approach



Main Idea

```
predicate uniform_nbh
(array[int] of var int: x,float: destrRate)

= forall(i in index_set(x))(
    uniform(0.0,1.0) > destrRate
      -> x[i] = sol(x[i])
);

constraint uniform_nbh([x1,x2,x3], 0.7);
```

Compile neighbourhood as constraint. Only once!

Our Approach

Push the NBH evaluation to the solver

- Keep the MiniZinc NBH definitions
- Add them as constraints to the model
- Use solver variables to represent state
- Hijack propagation to evaluate expressions
- Use restarts to re-evaluate the state

Main Idea

```
predicate uniform_nbh
(array[int] of var int: x,float: destrRate)

= forall(i in index_set(x))(
    uniform(0.0,1.0) > destrRate
      -> x[i] = sol(x[i])
);

constraint uniform_nbh([x1,x2,x3], 0.3);
```

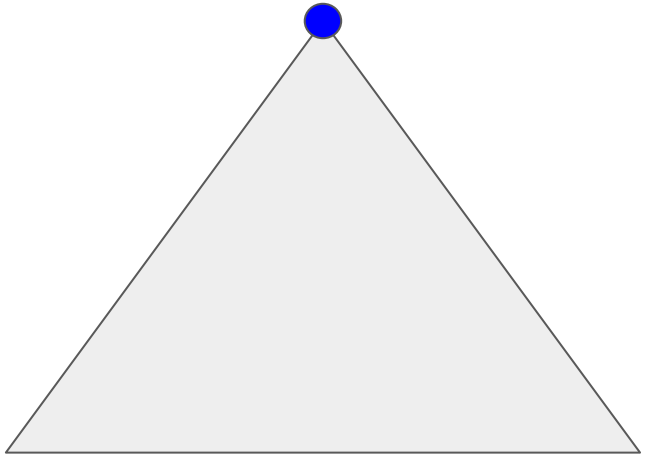
Compile neighbourhood as constraint. Only once!

Main Idea

```
cons. sol(x1,xi1)  cons. uniform(0.0,1.0,xi4)
cons. sol(x2,xi2)  cons. uniform(0.0,1.0,xi5)
cons. sol(x3,xi3)  cons. uniform(0.0,1.0,xi6)

    cons. (xi4 > 0.3) -> (x1 = xi1)
    cons. (xi5 > 0.3) -> (x2 = xi2)
    cons. (xi6 > 0.3) -> (x3 = xi3)
```

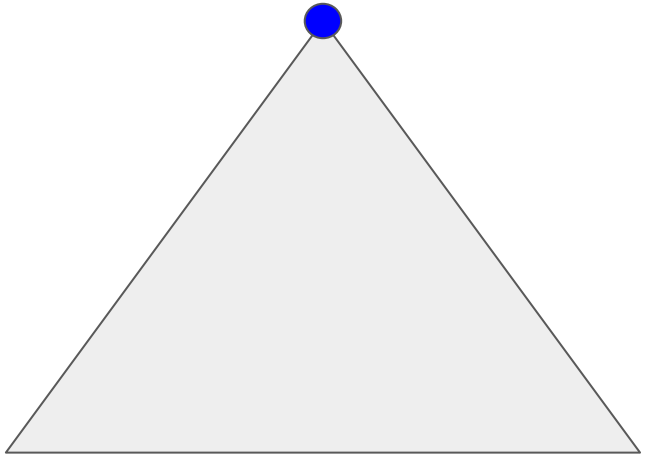
Main Idea

```
cons. (xi4 > 0.3) -> (x1 = xi1)
cons. (xi5 > 0.3) -> (x2 = xi2)
cons. (xi6 > 0.3) -> (x3 = xi3)
```



Main Idea

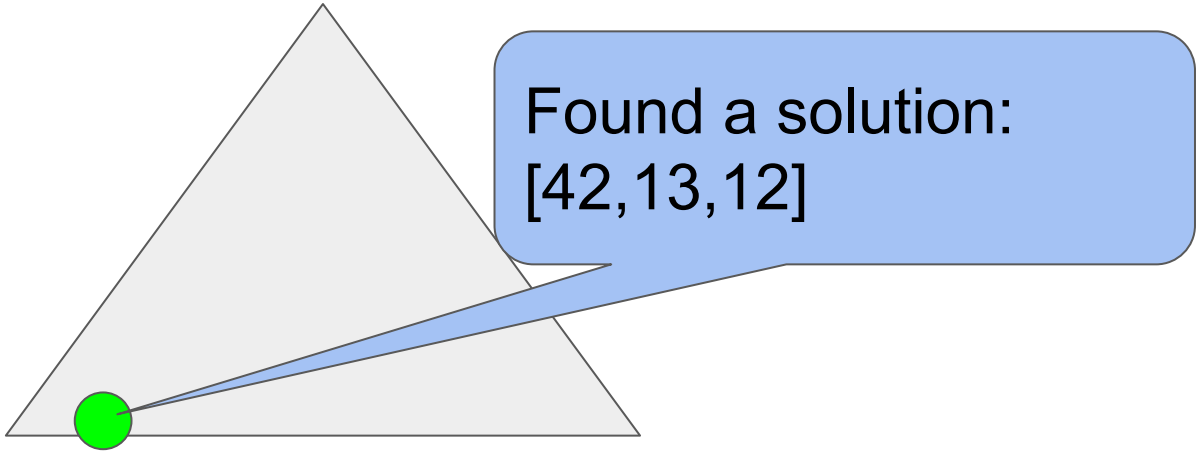
```
cons. (0.1 > 0.3) -> (x1 = xi1)
cons. (0.8 > 0.3) -> (x2 = xi2)
cons. (0.9 > 0.3) -> (x3 = xi3)
```



No solutions yet

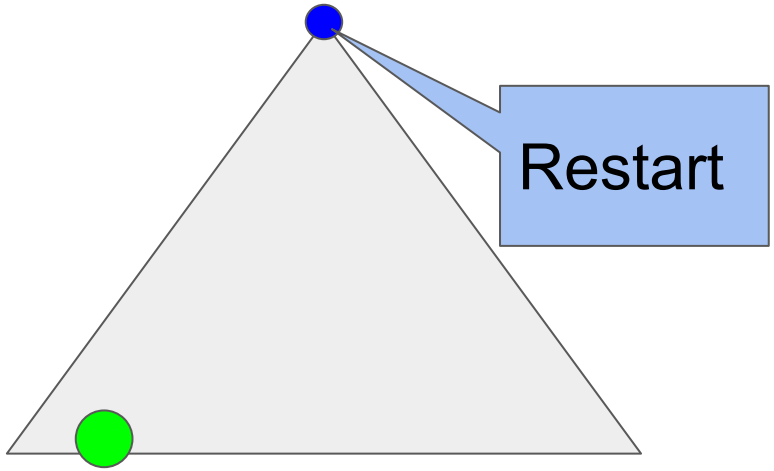
Main Idea

```
cons. (0.1 > 0.3) -> (x1 = xi1)
cons. (0.8 > 0.3) -> (x2 = xi2)
cons. (0.9 > 0.3) -> (x3 = xi3)
```



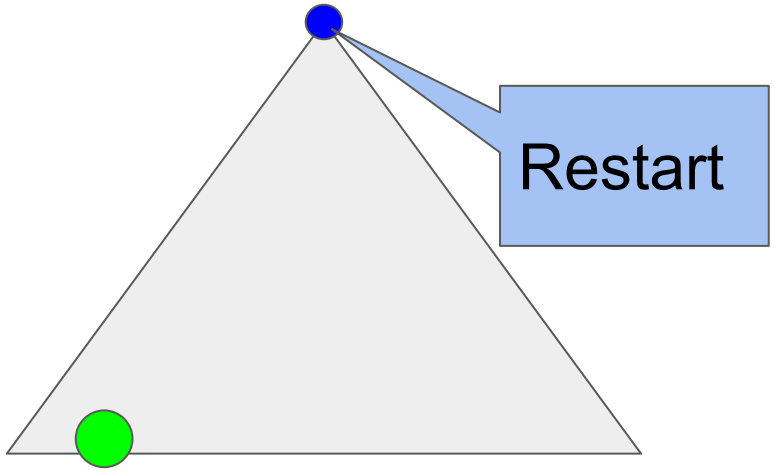
Main Idea

```
cons. (xi4 > 0.3) -> (x1 = xi1)
cons. (xi5 > 0.3) -> (x2 = xi2)
cons. (xi6 > 0.3) -> (x3 = xi3)
```



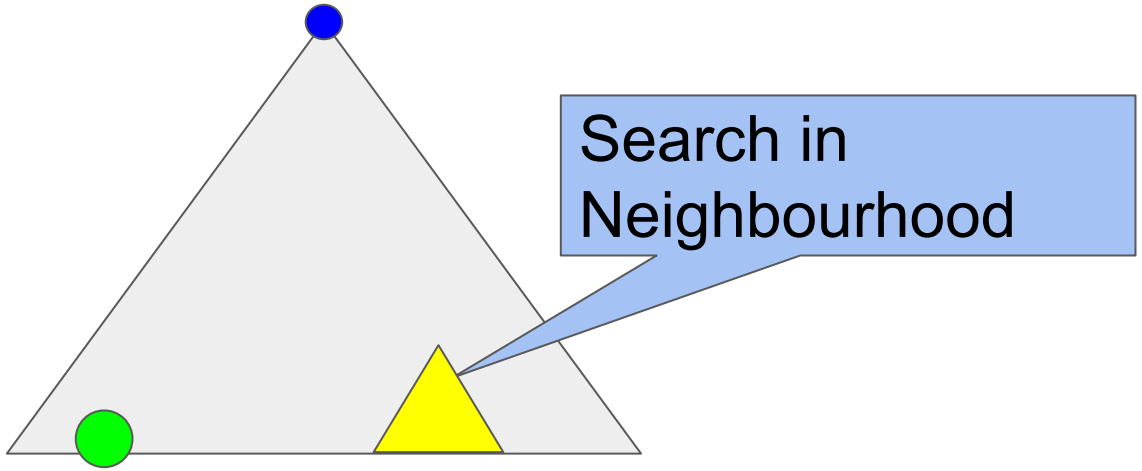
Main Idea

```
cons. (0.5 > 0.3) -> (x1 = 42)
cons. (0.2 > 0.3) -> (x2 = 13)
cons. (0.8 > 0.3) -> (x3 = 12)
```



Main Idea

```
cons. (0.5 > 0.3) -> (x1 = 42)
cons. (0.2 > 0.3) -> (x2 = 13)
cons. (0.8 > 0.3) -> (x3 = 12)
```



Solver Requirements

On restart propagate:

- `uniform(l,u,X)` ↦ `X` := random value from `l..u`
- `sol(X1,X2)` ↦ `X2` := last solution of `X1`
- `lastval(X1,X2)` ↦ `X2` := last assigned value of `X1`
- `status(X)` ↦ `X` := solver status (`START`,`SAT`,…)

Last Value Builtin

Consists of two parts:

- Apply latest value on restart
- Propagator that stores the last value
  - Activate when variable is fixed
  - Store value to global state

Example: Round Robin

```
predicate round_robin(array[int] of var bool: nbhs)
= let {
   int: N = length(nbhs);
   var -1..N-1: select;          % Neighbourhood selection
} in

if status()=START
   then select = -1
   else select = (lastval(select) + 1) mod N
endif

/\ forall(i in 1..N) (select=i-1 -> nbhs[i]);
```

Experiments

- Three models: GBAC, Steel Mill Slab, RCPSP-wet
- Show that the overhead is minimal
  - In Gecode 3% fewer nodes per second
- Show the usability in different solvers
  - CP solver: Gecode (110 lines of code)
  - LCG solver: Chuffed (126 lines of code)

Concluding remarks

- Easy to implement within a solver
- Minimal overhead
- More of MiniSearch could be compiled into FlatZinc

Job opportunity

- Three postdoctoral fellowships (3 years)
  - CP, MIP, SAT, SMT, programming languages
  - apply now
  - http://careers.pageuppeople.com/513/cw/en/job/580992/research-fellow-ai-optimisation-group
- Three continuing teaching and research academic positions
  - advertised within 2 weeks
- Come and join a very vibrant optimization group!

Thank you!

Jip J. Dekker

PhD Candidate @ Monash University
@ Data61, CSIRO

Improving meta-search in MiniZinc

Contact me:

jip.dekker@monash.edu

@DekkerOne

@Dekker1

Experiments: GBAC

	best known	gecode		gecode-fzn		gecode-replay		chuffed		chuffed-fzn	
Instance	min	$\int$	min	$\int$	min	$\int$	min	$\int$	min	$\int$	min
UD2-gbac	**146**	1502k	12515	93k	376^{16}	**92k**	$\mathbf{362}^{15}$	1494k	12344	**207k**	$\mathbf{598}^{54}$
UD4-gbac	**396**	1517k	12645	121k	$\mathbf{932}^{24}$	**120k**	$\mathbf{932}^{24}$	1151k	9267	**160k**	$\mathbf{1142}^{5}$
UD5-gbac	**222**	2765k	23028	283k	$\mathbf{2007}^{39}$	**281k**	$\mathbf{2007}^{39}$	2569k	21233	**483k**	$\mathbf{2572}^{22}$
UD8-gbac	**40**	1195k	9611	21k	$\mathbf{53}^{26}$	**20k**	$\mathbf{53}^{26}$	1173k	9559	**114k**	$\mathbf{76}^{26}$
reduced_UD4	**949**	629k	4917	**114k**	$\mathbf{950}^{0}$	**114k**	$\mathbf{950}^{0}$	715k	5491	**117k**	$\mathbf{950}^{0}$

Experiments: SMS

	best known	gecode		gecode-fzn		gecode-replay		chuffed		chuffed-fzn	
Instance	min	$\int$	min	$\int$	min	$\int$	min	$\int$	min	$\int$	min
bench_13_0	**0**	3247	27	20	$\mathbf{0}^0$	**19**	$\mathbf{0}^0$	1315	9	**50**	$\mathbf{0}^0$
bench_14_1	**0**	1248	**0**	32	$\mathbf{0}^0$	**31**	$\mathbf{0}^0$	**72**	**0**	79	$\mathbf{0}^0$
bench_15_11	**0**	4458	30	27	$\mathbf{0}^0$	**26**	$\mathbf{0}^0$	143	**0**	**65**	$\mathbf{0}^0$
bench_16_10	**0**	2446	**0**	**19**	$\mathbf{0}^0$	**19**	$\mathbf{0}^0$	122	**0**	**51**	$\mathbf{0}^0$
bench_19_5	**0**	3380	28	12	$\mathbf{0}^0$	**11**	$\mathbf{0}^0$	3040	19	**31**	$\mathbf{0}^0$

Experiments: RCPSP

	best known	gecode		gecode-fzn		gecode-replay		chuffed		chuffed-fzn	
Instance	min	$\int$	min	$\int$	min	$\int$	min	$\int$	min	$\int$	min
j30_1_3-wet	**93**	20k	161	**11k**	$\mathbf{93}^{0}$	**11k**	$\mathbf{93}^{0}$	**3k**	**93**	13k	$\mathbf{93}^{0}$
j30_43_10-wet	**121**	19k	158	15k	$\mathbf{121}^{0}$	**14k**	$\mathbf{121}^{0}$	**10k**	**121**	15k	$\mathbf{121}^{0}$
j60_19_6-wet	**227**	54k	441	**29k**	$\mathbf{235}^{3}$	**29k**	$\mathbf{235}^{3}$	63k	487	**29k**	$\mathbf{227}^{0}$
j60_28_3-wet	**266**	94k	770	**33k**	$\mathbf{273}^{0}$	**33k**	$\mathbf{273}^{0}$	79k	604	**35k**	$\mathbf{272}^{1}$
j90_48_4-wet	**513**	199k	1653	72k	$\mathbf{535}^{2}$	**71k**	$\mathbf{535}^{2}$	201k	1638	**109k**	$\mathbf{587}^{2}$