A Constraint Model for Google's Hash Code Optimizing a Data Center Problem (course 1DL441) Uppsala University – Autumn 2018 Project Report by Team 12

Diego Castillo Tristan Wright

12th February 2019

Preface

All experiments in the proceeding sections were run under Linux Ubuntu 18.04 (64 bit) on an Intel Xeon E5520 of 2.27 GHz, with 4 processors of 4 cores each, with a 24 GB RAM and an 8 MB L2 cache (a ThinLinc computer of the Uppsala University's IT department).

1 Optimizing a Data Center (ODC)

1.A Problem Definition

The ODC problem comes from the 2015 qualification round of the Google Hash competition. Given a data center schema, the goal is to place servers in slots of a row, and assign them to logical pools such that the minimum guaranteed capacity among all pools is maximized. This is naturally subject to a few constraints:

- Servers are placed in valid slots.
- Servers are logically divided into pools.
- Each server belongs to exactly one pool.
- The capacity of a pool is the capacity of the servers in that pool.
- The guaranteed capacity of pool i is $0 \leq i < P$ where P is the number of pools (zero indexed), and R is the number of rows is defined as:

$$\mathrm{gc}_i = \min_{0 \leq r < R} \Big(\sum_{k=0,\text{server } k \text{ in pool } i} c_i - \sum_{k=0,\text{ server } k \text{ in pool } i,\text{ server } k \text{ in row } r} c_i \Big)$$

The score (objective value) of a solution is then defined as:

$$\text{score} = \min_{0 \leq i < P} \mathrm{gc}_i$$

1.B Data Pre-processing

Figure 1 shows sample instance data for the problem, the first line indicates the number of rows, slots in each row, number of unavailable slots, pools, and servers - in that order. The next line specifies the location of the only unavailable slot in this instance, where the first column represents the row, and the second one the slot within that row (using zero-based index notation). Finally, the following five lines state in the first column the width of each available server, and the second column their capacities.

```
2 5 1 2 5
0 0
3 10
3 10
2 5
1 5
1 1
```

Figure 1: A sample problem's instance data given by Google.

The file containing the input data as shown in figure 1 is manually transformed into valid MiniZinc format as shown in figure 2. This is the data format which is used by the models shown in the following sections. Note that the parameter for the number of pools has been dropped from the array on the first line, this is so that the models can take the number of pools as input.

```
Env = [2, 5, 1, 5];
UnavailableSlots = [|0, 0|];
Server = [|
  3, 10 |
  3, 10 |
  2, 5  |
  1, 5  |
  1, 1  |];
```

Figure 2: Result after transforming the input data shown in figure 1 into valid MiniZinc format.

1.C Model A

Implementation

Listing 1 shows the full implementation of a model in MiniZinc to solve the ODC problem.

```
include "./data/dc.dzn";
include "globals.mzn";

%----Parameters----
% Initial Environment array
array[1..4] of int: Env;

% The number of logical pools
```

```
int: nPools;
set of int: Pools = 1..nPools;

% There are Env[1] number of rows
int: nRows = Env[1];
set of int: Rows = 1..nRows;

% There are Env[2] number of slots per row
int: nSlots = Env[2];
set of int: Slots = 1..(nSlots * nRows);

% There are Env[3] unavailable slots
int: nUnavailable = Env[3];
set of int: Unavailable = 1..nUnavailable;

% There are Env[4] available servers
int: nServers = Env[4];
set of int: Servers = 1..nServers;

% Unavailable[r,p] is an unavailable slot at row r and position p
  (zero indexed)
array[Unavailable, 1..2] of int: UnavailableSlots;

% Server[i,..] is the size and capacity of server i
array[Servers, 1..2] of int: Server;

%----Pre-computed parameters----
% Width[s] is the width of server s
array[Servers] of int: Width = [Server[s, 1] | s in Servers];

% Capacity[s] is the capacity of server s
array[Servers] of int: Capacity = [Server[s, 2] | s in Servers];

% Unavailable[u] is the 1d coordinate of an unavailable slot.
array[Unavailable] of int: UnusableSlot = [
  if UnavailableSlots[u, 1]  = 0 then
    UnavailableSlots[u, 2] + 1
  else
    (nSlots * UnavailableSlots[u, 1]) + UnavailableSlots[u, 2]
  endif
  | u in Unavailable];

% MaxWidth[s] is the maximum width that can be fit in slot s
set of int: Widths = 1..max(Server[..,1]);
array[Slots] of 0..max(Widths): MaxWidth = [card({w | w in Widths
  where
    if (s mod nSlots) = 0 then
      % Only servers of width 1 can fit in the last slot of any row
      w = 1
    else
      (s mod nSlots) + w <= nSlots + 1
    endif
    /\
    % No overlap with an unavailable slot
    forall(u in UnusableSlot)(not(u in (s..s+w-1)))
  }) | s in Slots];

% SlotRow[s] is the row of slot s
array[Slots] of var Rows: SlotRow = [
  if (s mod nSlots) = 0 then
    ((s div nSlots) mod nSlots)
  else
    ((s div nSlots) mod nSlots) + 1
  endif
  | s in Slots];

%----Decision variables----
% VWidth[s] is the width of server s, where a server of width=0
  means it's unused
array[Servers] of var 0..max(Width): VWidth;

% Slot[s] is the left-most slot for server s
array[Servers] of var Slots: Slot;

% Pool[p, s] is 1 if server s is in pool p
array[Pools, Servers] of var 0..1: Pool;

% TotalCapacity[p] is the total capacity of pool p
array[Pools] of var 0..max(Capacity) * nServers: TotalCapacity;

% RowCapacity[p, r] is the total capacity of pool p in row r
array[Pools, Rows] of var 0..max(Capacity) * nServers: RowCapacity;

% GuaranteedCapacity[p] is the guaranteed capacity of pool p
array[Pools] of var 0..max(Capacity): GuaranteedCapacity;

%----Constraints----
% No servers overlap over slots
constraint disjunctive(Slot, VWidth);

% Server's VWidth can be any of {0, Width[s]}
constraint forall(s in Servers)(
  VWidth[s] in {0, Width[s]}
);

% Servers are placed in valid positions
constraint forall(s in Servers)(
  Width[s] <= MaxWidth[Slot[s]]
```

```
);

% Each server of VWidth > 0 (used) is in exactly 1 pool, 0 otherwise
constraint forall(s in Servers)(
  count(Pool[..,s], 1) = bool2int(VWidth[s] > 0)
);

% Compute the total capacity of each pool
constraint forall(p in Pools)(
  TotalCapacity[p] = sum(s in Servers)(Pool[p, s] * Capacity[s])
);

% Compute total capacity of each pool per row
constraint forall(p in Pools, r in Rows)(
  RowCapacity[p, r] = sum(s in Servers)(bool2int(SlotRow[Slot[s]] =
    r) * Pool[p, s] * Capacity[s])
);

% Compute the guaranteed capacity of each pool
constraint forall(p in Pools)(
  GuaranteedCapacity[p] = TotalCapacity[p] - max(RowCapacity[p,..])
);

%----Symmetry Constraints----
% Break Pool 2d matrix row symmetry
constraint symmetry_breaking_constraint(
  forall(p in Pools diff {max(Pools)})
    (lex_greatereq(Pool[p,..],Pool[p+1,..])));

% Break Pool 2d matrix col symmetry
constraint symmetry_breaking_constraint(
  forall(s in Servers diff {max(Servers)})
    (lex_greatereq(Pool[..,s],Pool[..,s+1])));

%----Objective----
solve :: seq_search([
    int_search(VWidth, largest, indomain, complete),
    int_search(Slot, input_order, indomain, complete),
    int_search(array1d(Pool), input_order, indomain_max, complete)
  ])
  maximize min(GuaranteedCapacity);
```

Listing 1: Implementation of a model in MiniZinc to solve the ODC problem.

Description

Parameters. There are three parameters from the input file and several which are extrapolated from these. From the input file, `Env` is an one-dimensional array of four parameters: number of rows, number of slots, number of unusable slots, and number of servers. Four integer parameters are assigned from these and five set-of-integer parameters are created from those.

The second parameter, `UnavailableSlots`, from the file input is an `nUnavailable` by 2 array of unusable slots. Where the first column is the row and the second column is the slot number. These are slots in which servers cannot be placed. These are zero-based coordinates, which we fix when creating pre-computed parameters.

The final parameter, `Server`, is a `nServers` by 2 array of servers, where the first column is the server's width and the second is the server's capacity.

Pre-computed Parameters. There are five pre-computed parameters which make full use of MiniZinc's conditional and array comprehension syntax. `Width[s]` and `Capacity[s]` are the width and capacity of server `s` respectively. `UnusableSlot` is the one-dimensional coordinate of an unusable slot as computed from the parameter `UnavailableSlots`. `MaxWidth[s]` is the maximum width that can be fit in slot `s` such that a server does not overflow the length of a row or overlap with a slot in `UnusableSlot`. Finally, `SlotRow[s]` is the row of slot `s`.

Decision Variables. There are six decision variables. `VWidth` is the variable width of a server, a constraint later fixes each element to be the width of the server or, for the purposes of the `disjunctive` predicate, zero which means it is unused. `Slot` is the left most slot for server `s`, this is used in a constraint to make sure that the server is in a valid slot for its width. `Pool` is a binary matrix by `Pools` and `Servers` which is a 1 if a server belongs to that pool, 0 otherwise. `TotalCapacity` is the capacity of each pool. `RowCapacity` is a matrix of `Pools` and `Rows` which contains the total capacity of a pool in a row. `GuaranteedCapacity` is the guaranteed capacity of a pool.

Constraints. There are nine constraints, two of them are for breaking symmetry (to be discussed below). They can be separated into three categories, packing the servers into the rows, placing the servers into pools, and computing the guaranteed capacities of these placements. The `disjunctive` predicate is used to fit `VWidth` into the possible positions `Slot`. It is the primary constraint which "packs" the servers into the rows. This is possible by treating the servers as actions that need to be scheduled and with each server's width interpreted as its duration. Because of this, `VWidth` is constrained to be either the width of server `s` or zero. Next, a constraint ensures that the `Slot` for server `s` supports the width of it by using the pre-computed parameter `MaxWidth`. Finally, each server where `VWidth` is greater than zero (used) is constrained to belong to exactly one pool, none otherwise.

The rest of the decision variables are constrained based in the definition of the guaranteed capacity of a pool given by Google.

Redundant Constraints There are several redundant constraints in the problem, but ultimately none of them made it to the model.

Since the problem is trying to maximize the minimum guaranteed capacity among all pools, the more servers there are in the data center, the better these can be distributed. As a result, it is implied that each row in the data center must have servers:

```
    constraint redundant_constraint(forall(r in Rows)(sum(RowCapacity[..,r]) >
0));
```

Additionally, since the guaranteed capacity of a pool is the minimum guaranteed it has when at most one row is shutdown, any interesting solution (overall minimum guaranteed capacity larger than zero) will make sure each pool has servers in at least two distinct rows (such that when one goes down, the other one still works). This redundant constraint can be defined as:

```
    constraint redundant_constraint(forall(p in Pools)(count(RowCapacity[p,..],
0) <= nRows - 2));
```

The model was evaluated with and without these two redundant constraints, and no improvement was seen when using them. Hence, it was decided not to include them in the model for simplicity.

Symmetry-Breaking Constraints Two symmetry-breaking constraints were implemented:

```
% Break Pool 2d matrix row symmetry
constraint symmetry_breaking_constraint(
  forall(p in Pools diff {max(Pools)})
    (lex_greatereq(Pool[p,..],Pool[p+1,..])));

% Break Pool 2d matrix col symmetry
constraint symmetry_breaking_constraint(
  forall(s in Servers diff {max(Servers)})
    (lex_greatereq(Pool[..,s],Pool[..,s+1])));
```

These symmetry-breaking constraints were inspired by the agricultural problem studied in class. The idea is that pools are just things, and switching their order in the `Pool` decision variable produces the same results and should then be avoided to accelerate the search. The same happens column-wise; switching the order in which the servers within a pool are assigned produces the same solution and should be avoided.

Search Annotations. There are three search annotations which help finding solutions:

`int_search(VWidth, largest, indomain, complete)`

This will try set the width of each server by choosing larger servers in the domain and choosing ascending values. Choosing larger servers to be placed first makes scheduling faster by filling up the schedule (each row) as fast as possible.

`int_search(Slot, input_order, indomain, complete)`

This will fill slots according to their ordering in the input and assign ascending values to each one, in other words this fills slots sequentially using larger server ID's.

`int_search(array1d(Pool), input_order, indomain_max, complete)`

This annotation will assign pools in the order of their input to a maximum value. Remember that `Pool` is a binary matrix, so the `indomain_max` is 1. What this could lead to is a staggered looking matrix:

```
[| 1 1 0 0 |
   0 1 1 0 |
   0 0 1 1 |]
```

Compilation and Running Instructions.

Compiling and running `ODC.mzn` requires input for the parameter `nPools`. This is possible using the `-D` option in the command line or through a popup window in the MiniZinc IDE.

For any backend provided on your machine the model can be compiled and run for `nPools=5` by typing `minizinc ODC.mzn -D "nPools=5" --solver solver-id` where `solver-id` is replaced with the correct solver. Consult `minizinc --solvers` for the list of solvers available. Make sure the folder and file `data/dc.dzn` is in the same directory as `ODC-relaxed.mzn`

Sample Test-Run Command. To run a full test to replicate results below run:

```
~/minizinc2.2/run-solvers.sh -opt -m ODC.mzn \
-r "nPools 5 45 10" -t 1800000 -o results.tex
```

Evaluation

The results simultaneously demonstrate the power of the `fzn-oscar-cbls` search strategy as well as the insufficiency of the model despite such a generous timeout duration. `Gecode` gives all zeros, when developing the model locally it was quickly concluded that `Gecode` would not be the right solver for this model. `Chuffed` finds a solution for `nPools` = 5. `Gurobi` throws an error each time. `fzn-oscar-cbls` incredibly finds feasible solutions up to `nPools` = 25 and errors on 45. `Picat`, similar to `Gurobi` cannot even approach this probem, timing out on even the smallest solutions. Suffice to say, these scores would not qualify us to participate in the final round of the Google Hash Code competition [1].

Backend	Gecode (CP)		Chuffed (LCG)		Gurobi (MIP)		fzn-oscar-cbls (CBLS)		Picat (SAT)	
`nPools`	`obj`	time	`obj`	time	`obj`	time	`obj`	time	`obj`	time
5	0	t/o	18	t/o	ERR	–	22	t/o	–	t/o
15	0	t/o	0	t/o	ERR	–	19	t/o	–	t/o
25	0	t/o	0	t/o	ERR	–	9	t/o	–	t/o
35	0	t/o	0	t/o	ERR	–	0	t/o	–	t/o
45	0	t/o	0	t/o	ERR	–	ERR	–	–	t/o

Table 1: Results for the model in listing 1 run from `nPools` 5 to 45 with a step of 10. In the 'time' column, if the reported time is less than the time-out (1800000 milliseconds here), then the reported objective value in the '`obj`' column was the first solution; else the time-out is indicated by 't/o' and the reported objective value is either the best value found, but *not* proven optimal, before timing out, or '–', indicating that no feasible solution was found before timing out.

1.D Model B

Listing 2 shows a second model to solve the ODC problem using MiniZinc. It differs from the model in listing 1 in that it pre-computes a random pool for each server (whether used or not) in the range 1..nPools. The problem the model is trying to solve is slightly simplified to “given a particular pool configuration, what is the best way to schedule servers in the data center?”

Implementation

```
include "./data/dc.dzn";
include "globals.mzn";

%----Parameters----
% Initial Environment array
array[1..4] of int: Env;

% The number of logical pools
int: nPools;
set of int: Pools = 1..nPools;

% There are Env[1] number of rows
int: nRows = Env[1];
set of int: Rows = 1..nRows;

% There are Env[2] number of slots per row
int: nSlots = Env[2];
set of int: Slots = 1..(nSlots * nRows);

% There are Env[3] unavailable slots
int: nUnavailable = Env[3];
set of int: Unavailable = 1..nUnavailable;

% There are Env[4] available servers
int: nServers = Env[4];
set of int: Servers = 1..nServers;

% Unavailable[r,p] is an unavailable slot at row r and position p
  (zero indexed)
array[Unavailable, 1..2] of int: UnavailableSlots;

% Server[i,..] is the size and capacity of server i
array[Servers, 1..2] of int: Server;

%----Pre-computed parameters----
% Width[s] is the width of server s
array[Servers] of int: Width = [Server[s, 1] | s in Servers];

% Capacity[s] is the capacity of server s
array[Servers] of int: Capacity = [Server[s, 2] | s in Servers];

```

```
% Unavailable[u] is the 1d coordinate of an unavailable slot.
array[Unavailable] of int: UnusableSlot = [
  if UnavailableSlots[u, 1]  = 0 then
    UnavailableSlots[u, 2] + 1
  else
    (nSlots * UnavailableSlots[u, 1]) + UnavailableSlots[u, 2]
  endif
  | u in Unavailable];

% MaxWidth[s] is the maximum width that can be fit in slot s
set of int: Widths = 1..max(Server[..,1]);
array[Slots] of 0..max(Widths): MaxWidth = [card({w | w in Widths
  where
    if (s mod nSlots) = 0 then
      % Only servers of width 1 can fit in the last slot of any row
      w = 1
    else
      (s mod nSlots) + w <= nSlots + 1
    endif
    /\
    % No overlap with an unavailable slot
    forall(u in UnusableSlot)(not(u in (s..s+w-1)))
  }) | s in Slots];

% SlotRow[s] is the row of slot s
array[Slots] of var Rows: SlotRow = [
  if (s mod nSlots) = 0 then
    ((s div nSlots) mod nSlots)
  else
    ((s div nSlots) mod nSlots) + 1
  endif
  | s in Slots];

% Pool[s] is the pool of server s (randomly selected)
array[Servers] of var Pools: Pool = [uniform(1, nPools) | s in
  Servers];

%----Decision variables----
% VWidth[s] is the width of server s, where a server of width=0
  means it's unused
array[Servers] of var 0..max(Width): VWidth;

% Slot[s] is the left-most slot for server s
array[Servers] of var Slots: Slot;

% TotalCapacity[p] is the total capacity of pool p
array[Pools] of var 0..max(Capacity) * nServers: TotalCapacity;

% RowCapacity[p, r] is the total capacity of pool p in row r
```

```
array[Pools, Rows] of var 0..max(Capacity) * nServers: RowCapacity;

% GuaranteedCapacity[p] is the guaranteed capacity of pool p
array[Pools] of var 0..max(Capacity): GuaranteedCapacity;

%----Constraints----
% No servers overlap over slots
constraint disjunctive(Slot, VWidth);

% Server's VWidth can be any of {0, Width[s]}
constraint forall(s in Servers)(
  VWidth[s] in {0, Width[s]}
);

% Servers are placed in valid positions
constraint forall(s in Servers)(
  Width[s] <= MaxWidth[Slot[s]]
);

% Compute the total capacity of each pool
constraint forall(p in Pools)(
  TotalCapacity[p] = sum(s in Servers)(
    bool2int(VWidth[s] > 0) * bool2int(Pool[s] = p) * Capacity[s]
  )
);

% Compute total capacity of each pool per row
constraint forall(p in Pools, r in Rows)(
  RowCapacity[p, r] = sum(s in Servers)(
    bool2int(SlotRow[Slot[s]] = r) * bool2int(VWidth[s] > 0) *
      bool2int(Pool[s] = p) * Capacity[s]
  )
);

% Compute the guaranteed capacity of each pool
constraint forall(p in Pools)(
  GuaranteedCapacity[p] = TotalCapacity[p] - max(RowCapacity[p,..])
);

%----Objective----
solve :: seq_search([
    int_search(VWidth, largest, indomain, complete),
    int_search(Slot, input_order, indomain, complete)
  ])
  maximize min(GuaranteedCapacity);
```

Listing 2: Implementation of a second model in MiniZinc to solve the ODC problem. This model pre-computes a random pool configuration, and attempts to find the best server scheduling such that the minimum guaranteed capacity of the data center is maximized.

Description

Parameters. All parameters stay as explained in the previous model.

Pre-computed Parameters. Except for the addition of `Pool`, all other pre-computed parameters are the same as in the previous model. The `Pool` is created by using the 'builtin' function `uniform`, which returns a sample from the uniform distribution defined by the lower-bound and upper-bound of its arguments, in this case 1..nPools:

```
% Pool[s] is the pool of server s (randomly selected)
array[Servers] of var Pools: Pool = [uniform(1, nPools) | s in
  Servers];
```

Decision Variables. Since it has been fixed during pre-computation, there is no `Pool` decision variable anymore. All other decision variables are the same as previously explained.

Constraints. As the pool is now pre-computed, the constraint which previously required each used server to belong to a pool is dropped. Most of the other constraints stay the same as before, but since every server now belongs to a pool (whether used or not), a few changes need to be implemented.

The constraint which defines the `TotalCapacity` of each pool is modified to first check if a server is being used by verifying if its `VWidth` is greater than zero, and then whether the pool it has been assigned to is the current pool:

```
% Compute the total capacity of each pool
constraint forall(p in Pools)(
  TotalCapacity[p] = sum(s in Servers)(
    bool2int(VWidth[s] > 0) * bool2int(Pool[s] = p) * Capacity[s]
  )
);
```

The same idea is applied to the constraint which defines `RowCapacity`:

```
% Compute total capacity of each pool per row
constraint forall(p in Pools, r in Rows)(
  RowCapacity[p, r] = sum(s in Servers)(
    bool2int(SlotRow[Slot[s]] = r) * bool2int(VWidth[s] > 0) *
     bool2int(Pool[s] = p) * Capacity[s]
  )
);
```

Redundant Constraints This model was also evaluated using the redundant constraint which says that each row must have servers on it. Similar to what happened with the previous model, no improvements were seen when using it, so it was decided not to include it for simplicity.

Since the pool configuration is fixed, the second redundant constraint does not apply to this model. Nonetheless, a better random initialization could have been implemented (e.g. pseudo-randomly initialize what pool each server belongs to such that the capacity of the servers is equally distributed among the pools).

Symmetry Breaking Constraints The model in listing 2 does not specify any symmetry-breaking constraints, but that does not mean there is no symmetry on it. One of the symmetries present is that switching the position of a server within the same row does not make a difference, since the pools are fixed. In this model, switching the position of a server only makes a difference if the new position where it is schedule at is in a different row. We failed to implement this symmetry breaking constraint as the view point from which our model is written makes it difficult to know over what rows a pool has servers in.

Search Annotations. This model uses two of the same search annotations as Model A for the `VWidth` and `Slot` decision variables. The search annotation on `Pool` is dropped because it is being precomputed.

Compilation and Running Instructions.

The methods for running and testing `ODC-fixed.mzn` are the same as for Model A in section 1.C.

Evaluation

The results are a large improvement to those in section 1.C. `Gecode` produces solutions for every instance of the dataset. `Chuffed` follows suite, generating consistently higher, and the best score, over all other backends. `Gurobi` errors twice but mysteriously approaches `nPools` = 25 and 35 only to timeout. `fzn-oscar-cbls` produces results for every instance as well but with a descending pattern. `Picat` has the same results as before, all timeouts. These results are heavily dependent on the specific initialization of `Pool`. These results may very well be the best minimum guaranteed capacities for that specific pool layout generated from `nPools`. A pigeon pecking on a large number pad with a range of 1 to 45 may –randomly– be able to create a pool that is better than the one we initialized here. These scores would also not qualify us to participate in the final round of the Google Hash Code competition [1]. Our best score of 95 on the full instance size would be tied at 222nd place.

Backend	Gecode (CP)		Chuffed (LCG)		Gurobi (MIP)		fzn-oscar-cbls (CBLS)		Picat (SAT)	
`nPools`	`obj`	time	`obj`	time	`obj`	time	`obj`	time	`obj`	time
5	99	t/o	99	t/o	ERR	–	98	t/o	–	t/o
15	94	t/o	98	t/o	ERR	–	87	t/o	–	t/o
25	95	t/o	97	t/o	–	t/o	84	t/o	–	t/o
35	92	t/o	96	t/o	–	t/o	59	t/o	–	t/o
45	85	t/o	95	t/o	ERR	–	50	t/o	–	t/o

Table 2: Results for the model in listing 2 run from `nPools` 5 to 45 with a step of 10. In the 'time' column, if the reported time is less than the time-out (1800000 milliseconds here), then the reported objective value in the '`obj`' column was the first solution; else the time-out is indicated by 't/o' and the reported objective value is either the best value found, but *not* proven optimal, before timing out, or '–', indicating that no feasible solution was found before timing out.

2 Conclusions

On fixed pools. Configuring a fixed pool is unlikely to solve the problem unless the modelers encounter an extraordinary stroke of luck or have available a large flock of pigeons as in section 1.D. It must be asked if there is a better way than simply randomly exploring pools. Consider an intuitive perspective on packing servers into pools. With two rows and five slots, and where each server has a width of 1 and all equal capacities, what is the best way to divide these servers into five pools? By columns of course, two servers per pool.

Now when server widths vary there will be fewer servers we can place however the intuitive strategy for pooling servers can still be used, go down each column and allocate the servers in the column to a pool. A difference being that there needs to be some staggering how we allocate them. In general, what is a method to make sure there is an *about* equal number of servers per pool?

Solutions following the intuitive method were drawn on paper but could not be converted to a model.

On scheduling versus packing. Ultimately, it seems as if scheduling was not the best viewpoint from which to model this problem. The thing is, scheduling cares about order. It searches for solutions in a per slot basis, which is a much more fine-grained search than required. All the problem asks for is on what rows are the servers, not what slots. This becomes self-evident in our model in the constraint that computes the `RowCapacity`, where the slot of a server is only used to channel the row where it is.

Additionally, unless handled with a symmetry-breaking constraint, scheduling is likely to try to change the position of a server within a row without changing its pool, which makes no difference at all in terms of the objective value.

Packing on the other hand, solves the problem from a much higher abstraction, by simply assigning servers to bins (rows in this case), without explicitly specifying the slot a server is in a row. As a result, it will not attempt to change the order in which servers are packed into a bin, because it does not make a difference (the load of the bin is the same).

Feedback to the Teachers

We felt constrained after the presentations that we could not really adapt to some bin packing perspective without somewhat plagiarizing the viewpoint and model from the opposing team. Inefficiencies in our approach became more and more apparent as the week wore on and we felt that we could not model our way out of.

This was a really fun problem to think about, but very hard to model when it came time to get to brass tacks. Would not recommend this problem to anyone next year.

References

[1] Google France. Results - Hash Code 2015. `https://sites.google.com/site/hashcode2015/results`, 2015. Online, Accessed October 25, 2018.