The Car Sequencing Problem

“In this problem, a car assembly line is set up to build cars on a production line divided into cells. There are five cells that install options on the cars, requiring different times to perform the operation for each cell, defined as the number of cars they can process in a period of time. There are seven different cars, each requiring a different set of options.

The production plan specifies the quantity of each car to build. The objective of the model is to compute a sequence of cars that the cells can process while minimizing the number of empty cars to insert to respect the load of the cells.”

To load this project, search for car in the project loader.

Formulation 1

Now we explain the model in detail.

Define number of car types, number of options and number of slots.

```
int  nbCars    = ...; // # of cars
int   nbOptions = ...;// # of options

int   nbSlots   = ...;// # of slots


range   Cars     = 1..nbCars;
range   Options  = 1..nbOptions;
range   Slots    = 1..nbSlots;
```

Load demand for each car type, and option needed for each type.

```
int demand[Cars] = ...;

int option[Options,Cars] = ...;
```

Define capacity constraint for each option. This means in a rolling window of width u, there can be no more than l cars needing this option.

```
tuple Tcapacity {
  int l;
  int u;
};

Tcapacity capacity[Options] = ...;
```

Calculate total option demand for each option.

```
int optionDemand[i in Options] = sum(j in Cars) demand[j] * option[i,j];
```

Define decision variables slot – which car type is produced; Define decision variable setup to indicate if a particular option is installed at each slot.

```
dvar int slot[Slots] in Cars;

dvar int setup[Options,Slots] in 0..1;
```

Define constraints that demand for each car type must be met.

```
  forall(c in Cars )

    sum(s in Slots ) (slot[s] == c) == demand[c];
```

Define the capacity constraints: for a rolling window, the capacity limit must be satisfied.

```
  forall(o in Options, s in 1..(nbSlots - capacity[o].u + 1) )
    sum(j in s..(s + capacity[o].u - 1)) setup[o,j] <= capacity[o].l;
```

Define constraints that setup for option o at slot s is 1 if it is required.

```
  forall(o in Options, s in Slots )
   setup[o,s] == option[o][slot[s]];
```

Define strengthening constraints that for each option, the minimum number of setup needed before a certain number of non-overlapping windows with width u is total option demand minus maximum number of options can be installed in these windows.

```
  forall(o in Options, i in 1..optionDemand[o])
```

```
    sum(s in 1 .. (nbSlots - i * capacity[o].u)) setup[o,s] >=

    optionDemand[o] - i * capacity[o].l;
```

Formulation 2

In this alternative formulation, we allow empty slots so that there is more tolerance for the capacity constraints.

Define the number, and range of configurations, options and demand along with capacity parameters.

```
int nbConfs   = ...;
int nbOptions = ...;
range Confs = 1..nbConfs;
range Options = 1..nbOptions;
int demand[Confs] = ...;
tuple CapacitatedWindow {
  int l;
  int u;
};

CapacitatedWindow capacity[Options] = ...;
```

Add configuration zero to represent empty slot. For a configuration zero, there is no option installed.

```
range AllConfs = 0..nbConfs;

int nbCars = sum (c in Confs) demand[c];
```

Add extra slots to relax the capacity constraints.

```
int nbSlots = ftoi(floor(nbCars * 1.1 + 5));
```

Define option parameter for each configuration.

```
int allOptions[o in Options, c in AllConfs] = (c == 0) ? 0 : option[o][c];
```

Define decision variables for each slot.

```
dvar int slot[Slots] in AllConfs;
```

Define decision variable for the last non-zero slot.

```
dvar int lastSlot in nbCars..nbSlots;
```

Objective is minimise the last scheduled slot (non-zero slot).

```
minimize lastSlot - nbCars; // Try to get to zero meaning all blanks at the end
```

Define constraints that number of slots scheduled for each configuration meets the demand.

```
 forall (c in Confs)
   count(slot, c) == demand[c];
```

Define constraints that rolling window constriants are met.

```
 forall(o in Options, s in Slots : s + capacity[o].u - 1 <= nbSlots)
   sum(j in s .. s + capacity[o].u - 1) allOptions[o][slot[j]] <= capacity[o].l;
```

Define constraints that last non-zero slot must have a valid configuration.

```
slot[lastSlot] > 0;
```

Define constraints that all slots after the last non-zero slot have configuration zero.

```
 forall (s in nbCars + 1 .. nbSlots)
   (s > lastSlot) => slot[s] == 0;
```