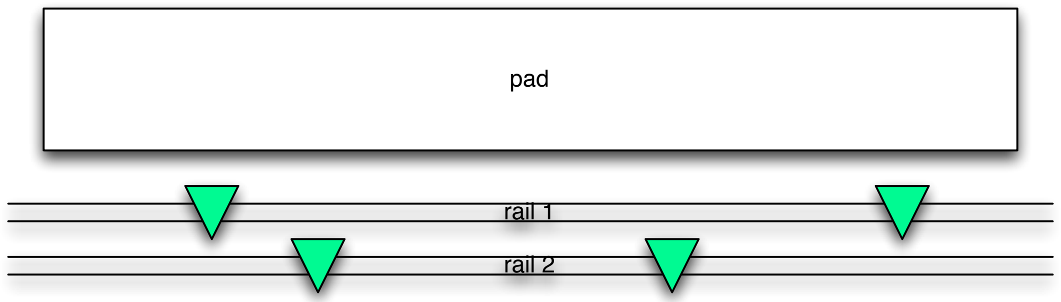
Constraint Programming Assignment: **Port Scheduling**

1 Problem Statement

Scheduling bulk mineral ports is a complex and important task for Australia. We have some of the busiest bulk mineral ports in the world. Ships arrive at the port to pick up a set of stockpiles. The aim is to delay the ships the least amount of time possible from when they arrive.

The stockpiles are stored on pads, which are very long. Here we consider the simple example of a single long pad. The stockpiles are stacked onto the pads over time, as trains bring in the materials for the stockpile. In this simple model we will just assume a stacking time for each stockpile and otherwise not worry about stacking. The pad causes a packing problem, we have to reserve space on the pad for a stockpile from the moment we start stacking to the moment it is fully reclaimed. Two stockpiles cant co-exist in space time.

The critical issue is the reclaiming of the stockpile from the pad to load it onto the ship. We will schedule the reclaimers.



Reclaimers run on rails beside the pads. There can be multiple parallel rails, and multiple reclaimers on each rail. Reclaimers on the same rail can not pass each other, they have to stay in the same order. Reclaimers on separate rails are free to move past each other.

When a ship arrives in berth we can start reclaiming its stockpiles. The reclaimer has to be at the westernmost end of the stockpile to reclaims the material, once finished it can move to the next task. The time to reclaim is given by the size of the stockpile. The reclaimer takes time to get to the stockpile to begin the reclaim.

The assignment is broken into stages though each of them uses the same input data format.

1.1 Stage A: Packing the Pad

The core of the port scheduling problem is to put stockpiles on the pad and then take them off again onto ships. This is a two dimensional packing problems where stockpiles are rectangles which have one dimension the segment of the pad they occupy from the westend to the eastend, and the second dimension is the time segment they occupy from the start of stacking to the end of reclaiming.

In the first stage you simply need to pack the stockpiles on the pad so they dont overlap in space-time.

Data is of the form:

```
int: nr; % number of reclaimers
         % number of rails = nr for stages A,B,C,D,E
         % number of rails = (nr + 1) div 2 for stage F
set of int: RECLAIMER = 1..nr;
bool: stageF;

int: ns; % number of stockpiles
set of int: STOCKPILE = 1..ns;
array[STOCKPILE] of int: size; % size in 10000 tonnes

int: maxtime; % time considered
set of int: TIME = 0..maxtime;

int: nsh; % number of ships
set of int: SHIP = 1..ns;
array[STOCKPILE] of SHIP: ship; % which ship takes stockpile
array[SHIP] of TIME: arrival; % when ship arrives in port

int: len; % length of pad
set of int: POSITION = 0..len;

int: stack_time;      % time to stack 1 unit (10000 tonnes)
int: reclaim_time;    % time to reclaim 1 unit
int: reclaim_speed;   % time for reclaimer to move 1 unit
```

For the purposes of this stage we only are interested in the stockpile data. You can assume the `stageF` parameter is `false` (until stage F).

The critical decisions that need to be made are: for each stockpile the position of its western most end on the pad, when it is started to stack, and when it is started to reclaim. In later stages a critical decision will be which reclaimer is used for each stockpile, but for this stage this can always be set to 1.

```
array[STOCKPILE] of var POSITION: westend;
array[STOCKPILE] of var TIME: stack;
array[STOCKPILE] of var TIME: reclaim;
array[STOCKPILE] of var RECLAIMER: which;
```

A stockpile occupies a position on the pad from its westend to its eastend = westend + size. It occupies a time on the pad from its stacking start time to the end of its reclaiming time. No two stockpiles can overlap in both time and position (but note that a stockpile could be in position 0..4 and another in position 4..8 at the same time and this is not an overlap, or on the pad from time 0..10, and then another stockpile in the same position from time 10..14 and this is not an overlap).

We assume that stacking a stockpile requires `stack_time` times its size in time units, and stacking must be finished before reclaiming begins. The *dwell time* of the stockpile is the period in between when stacking ends and reclaiming begins.

Reclaiming a stockpile requires `reclaim_time` times its size.

Output of the plan should be of the form

```
westend = array of positions of westend;
eastend = array of positions of eastend;
stack = array of stacking start times;
endstack = array of stacking end times;
reclaim = array of reclaiming start times;
finish = array of reclaiming end times;
which = array of which reclaimers used;
```

For stage A, the `which` array can be set to all 1s.

Note that you may well have to explore some complex search strategies to generate solutions to even this first stage version of the problem. Default search is not likely to be effective.

1.2 Stage B: Reclaimer Assignment

In reality each stockpile has to be reclaimed by one of the reclaimers in the port. For stage B we need to ensure that reclaiming is possible.

You should add constraints to your model to enforce that two stockpiles reclaimed by the same reclaimer do not overlap in time. The `which` variables are now important.

The input and output format is unchanged from stage A, but the `which` array has meaning.

1.3 Stage C: Ship Constraints

Each stockpile must be reclaimed onto the ship it is destined for. In this stage for the first time we consider the ship information.

You should add constraints to your model so that no stockpile can be reclaimed onto a ship before the arrival time of the ship. Also no two stockpiles can be reclaimed onto the same ship at the same time. Make sure these reclaims do not overlap in time.

1.4 Stage D: Reclaimer Movement

The present model assumes that reclaimers can instantly move from one position on the pad to another. This is not correct, we need to take into account its speed of movement.

Add constraints to your model to ensure that if a reclaimer finishes reclaiming a stockpile with westend at x and then has to start reclaiming a stockpile with westend at y there is at least $|y - x| * \texttt{reclaim\_speed}$ time between these two events.

1.5 Stage E: Objective

Now we are ready to consider the objective. The aim of the port scheduling is to minimize the total time of ships at berth. A ship is at berth from its arrival time to the end of the reclaiming of the last stockpile destined for this ship.

Add a definition of the objective to your model and change the model to minimize this value.

You may well need to significantly change your search strategy to get good solutions for the objective.

1.6 Stage F: Rail Constraints

In reality usually there are two reclaimers per rail line, and they cannot cross.

Add constraints in your model so that the two reclaimers on rail i numbered $2i-1$ and $2i$ for $i \in 1..nr\ div\ 2$ remain so the western one $2i-1$ is never east of the eastern one $2i$. Note they can legitimately be in the same position (this is for simplicity, its not very real). Note that if there are an odd number of reclaimers the last reclaimer is on its own rail and has no further constraints.

1.7 Visualization

In order to help you visualize the results of the schedule we provide a program: `portschedule_draw.cpp` which will take the output of the model and plot it. For example for the input data

```
nr = 2;
stageF = true;
ns = 7;
nsh = 4;
size = [4, 8, 2, 5, 4, 3, 3];
ship = [1,2,3,1,4,2,3];
maxtime = 200;
arrival = [ 0, 0, 20, 40 ];
len = 12;
stack_time = 4;
reclaim_time = 3;
reclaim_speed = 1;
```

gives a solution

```
westend = [0, 0, 4, 0, 8, 6, 8];
eastend = [4, 8, 6, 5, 12, 9, 11];
stack = [0, 29, 0, 85, 21, 0, 52];
endstack = [16, 61, 8, 105, 37, 12, 64];
reclaim = [16, 61, 23, 105, 40, 12, 128];
finish = [28, 85, 29, 120, 52, 21, 137];
which = [1, 1, 2, 1, 2, 2, 1];
```

The plot for this solution is shown in Figure 1 which shows for each the stockpile the stacking time, the dwell time the reclaiming time, and the path of each reclaimer in its reclaiming tasks. Note how the reclaimer 2 stays east (above) reclaimer 1, and we can see delays caused by moving the reclaimers. Be aware that sometimes the visualization can show a crossing which does not need to be there since it simply connects the reclaim jobs in order. The grader will detect when the crossing must occur and flag an error (for stage F).

2 Instructions

Edit `portschedule.mzn` to solve the optimization problem described above. Your `portschedule.mzn` implementation can be tested on the data files provided. In the MiniZincIDE use the *play* icon to test your model locally. At the command line use,

```
mzn-gecode ./portschedule.mzn ./data/<inputFileName>
```

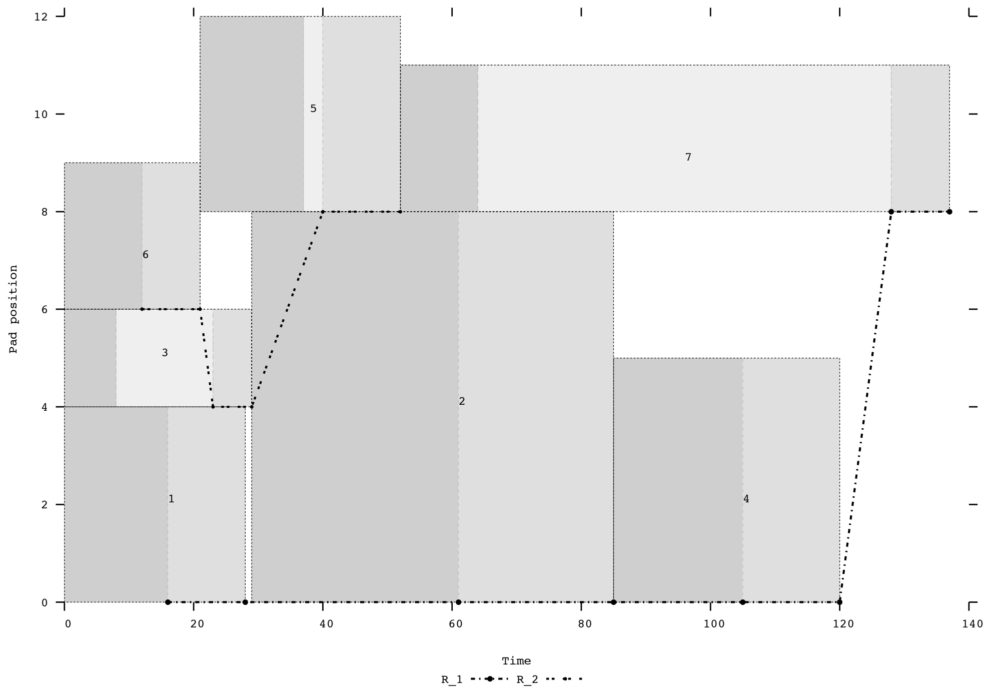


Figure 1: Pad diagram showing stacking, dwell, and reclaiming for each stockpile, as well as the path of the reclaimers.

to test locally. In both cases, your model is compiled with MiniZinc and then solved with the GECODE solver. The other method is to use the built-in `run+check` (available when the checker file `mzc` is also opened.

You will find several problem instances in the `data` directory.