wrong course name

Basic and Advanced Modelling for Discrete Optimisation Uppsala University – Autumn 2018 Project Report

Huu-Phuc Vo

27th September 2018

Streaming Videos Problem

All experiments were run under Linux Ubuntu 16.04 (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 IT department).

A Introduction

Nowadays, watching videos online is pervasive, especially watching videos from Youtube. When streaming videos from Youtube to a huge amount of people, who could be in the same city or from different continents, minimising the waiting time for all requests from clients are critical. In the context of the *Streaming videos* problem, the video-serving infrastructure includes (§1) remote data centers locating in thousands of kilometers away, (§2) cache servers, which store copies of popular videos, and (§3) endpoints, which each of them represents a group of users connecting to the Internet in the same geographical area. The expected solution for the *Streaming videos* problem is to decide which videos to put in which cache servers. The specification of the problem could be found in detailed at [1], and the data could be found at [2].

Task. Given a description of cache servers, network endpoints and videos, along with predicted requests for individual videos, the task is to *decide which videos to put in which cache servers* in order to *minimise* the average *waiting time* for all requests. In other word, the task is to *maximise* the average *saving time* for all given requests.

A.1 Problem description

Figure 1 illustrates the video serving network, which includes the data center, cache servers, and endpoints. The **data center** stores *all videos*. The size of video, the maximum capacity of cache servers are in megabytes (MB). Each **video** can be put in 0, 1, or more *cache servers*. Each **cache server** has a maximum capacity. Every **endpoint** is connected to the data center, however, it may be connected to 0, 1 or more cache servers. Each endpoint is characterised by the latency of its connection to the data center, and by the latencies to each cache server that it is connected to. The **predicted requests** provide data on how many times a particular video is requested from a particular endpoint.

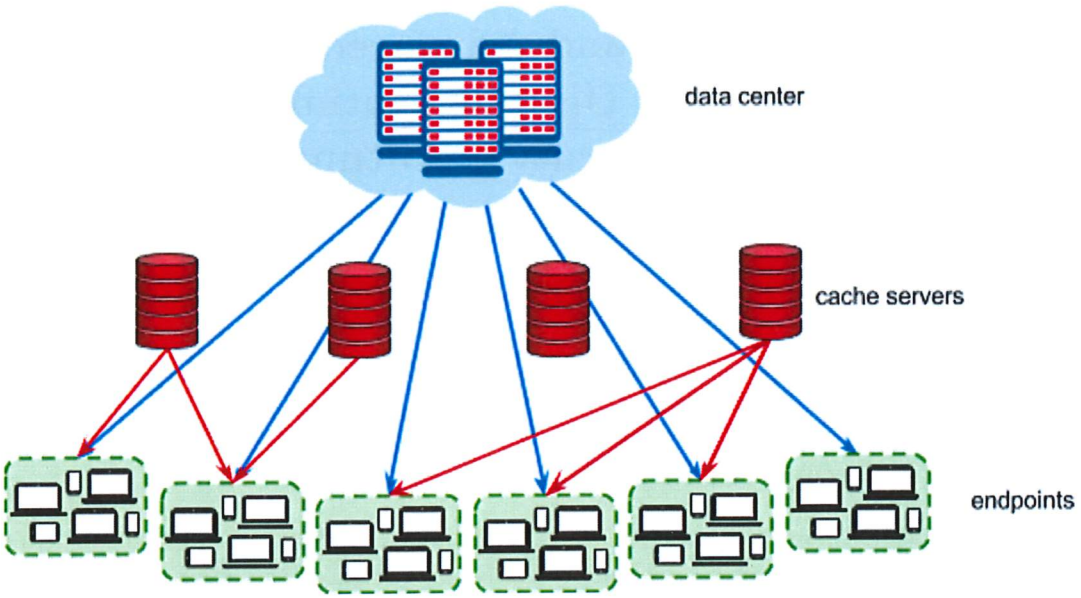

Figure 1: The video serving network

A.2 Example of input

Table 1 illustrates the input file. The original input data is given in the format that is not the instance for *MiniZinc*. Consequently, pre-processing the original inputs to *MiniZinc*'s instances is taken in the first place.

5 2 4 3 100	5 videos, 2 endpoints, 4 request descriptions, 3 caches 100MB each.
50 50 80 30 110	Videos 0, 1, 2, 3, 4 have sizes 50MB, 50MB, 80MB, 30MB, 110MB.
1000 3	Endpoint 0 has 1000ms datacenter latency and is connected to 3 caches:
0 100	The latency (of endpoint 0) to cache 0 is 100ms.
2 200	The latency (of endpoint 0) to cache 2 is 200ms.
1 300	The latency (of endpoint 0) to cache 1 is 300ms.
500 0	Endpoint 1 has 500ms datacenter latency and is not connected to a cache.
3 0 1500	1500 requests for video 3 coming from endpoint 0.
0 1 1000	1000 requests for video 0 coming from endpoint 1.
4 0 500	500 requests for video 4 coming from endpoint 0.
1 0 1000	1000 requests for video 1 coming from endpoint 0.

Table 1: Example of input file.

Firgure 2 shows the corresponding *MiniZinc*'s instance from input data 1.

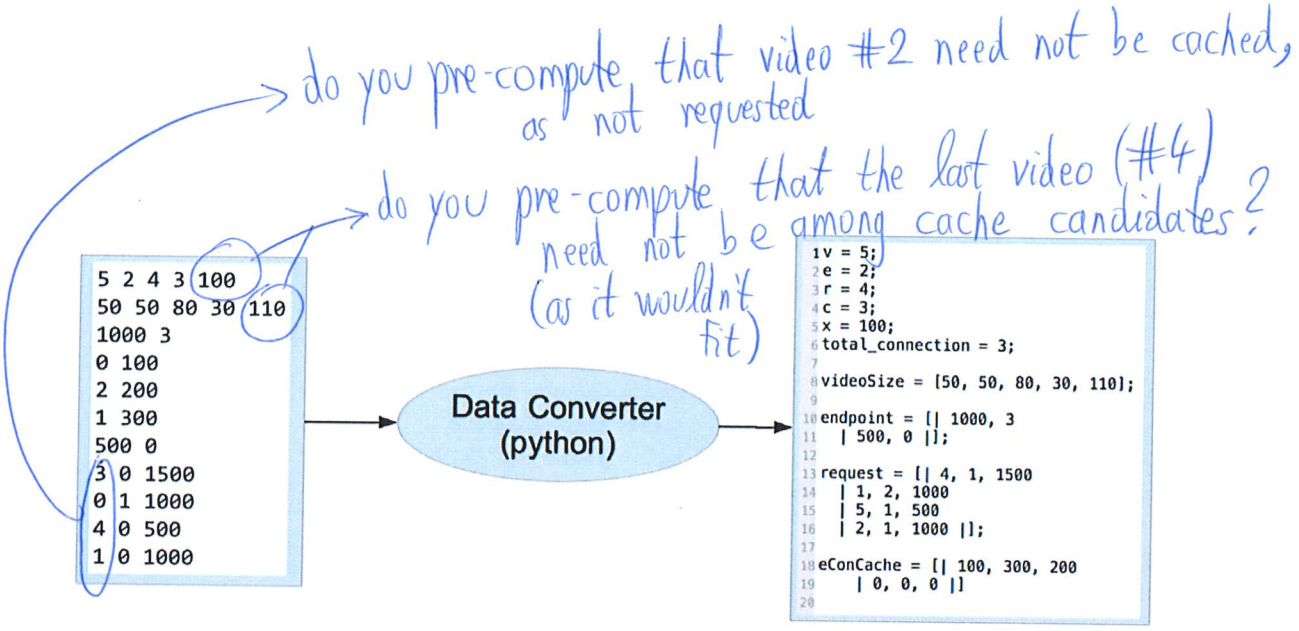

Figure 2: Convert original input data to *MiniZinc* instance.

A.3 Example of output

Table 2 illustrates the output file. Notice that the output example is not optimal solution. A better solution is to put videos 1 and 3 in cache 0 to maximise the saving time since the latency of cache 0 is minimal.

3	We are using all 3 cache servers.
0 2	Cache server 0 contains only video 2.
1 3 1	Cache server 1 contains videos 3 and 1.
2 0 1	Cache server 2 contains videos 0 and 1.

Table 2: Example of output file.

B Model

Listing 1 shows the model of streaming videos problem including model comments.

Listing 1: A *MiniZinc* model for the Streaming Videos problem

```
1 int: v;% (1 .. 10.000) the number of videos
2 int: e;% (1 .. 1.000) the number of endpoints
3 int: r;% (1 .. 1.000.000) the number of request descriptions
4 int: c;% (1 .. 1.000) the number of cache servers
5 int: x;% (1 .. 500.000) the capacity of each cache server in MB
6
7 set of int: VID  = 1..v;
8 set of int: VID0 = 0..v-1;
9
10 set of int: ENDPOINT = 1..e;
11 set of int: REQUEST = 1..r;
12
13 set of int: CACHE  = 1..c;
14 set of int: CACHE0 = 0..c;
15
16 set of int: CAPACITY = 1..x;
17 set of int: SIZE = 1..1000;
```

```
18
19 % sum of all connected caches K
20 int: total_connection;
21
22 set of int: CONN = 1..total_connection;
23
24 % video sizes
25 array[VID] of SIZE: videoSize;
26
27 % Ld: data center latency        ( 2 .. 4.000 )
28 % K: number of connected caches  ( 0 .. C )
29 enum LDK = {Ld, K};
30 array[ENDPOINT, LDK] of 0..4000: endpoint;
31
32 % Rv: requested video            ( 0 .. V )
33 % Re: coming from endpoint       ( 0 .. E )
34 % Rn: number of requests         ( 0 .. 10.000)
35 enum RVEN = {Rv, Re, Rn};
36 array[REQUEST, RVEN] of 0..10000: request;
37
38 % number of cache servers ( 0 .. C : 1.000)
39 var CACHE0: n;
40
41 % ei: from endpoint
42 % ci: to cache ( 0 .. C ),
43 % Lc: latency  ( 1 .. 500 < Ld ) in milliseconds
44 % eConCache[en, ca] :
45 %    Lc: connects to cache id;
46 %    0 : not connect
47 array[ENDPOINT, CACHE] of 0..1000: eConCache;
48
49 % 1: stored in data center
50 % 0: not in dc
51 array[ENDPOINT, VID] of var {0,1}: vInDc;
52
53 % ci : id of cache server being described
54 %      ( 0 .. C : 1.000 ), 0 : data center
55 % vi : id of videos stored in this cache server
56 %      ( 0 .. V : 10.000 )
57 array[CACHE, VID] of var {0,1} : usedCache;
58
59 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
60 % TASK
61 %   - decide which videos to put in which cache server
62 %   - minimize average waiting time for all requests
63 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
64
65 % P1: Precomputation: number of used caches
66 constraint n = sum(ca in CACHE)(
```

```
67      bool2int(exists(vi in VID)(usedCache[ca, vi] = 1))
68  );
69
70  % C1: Channelling constraint: Mark which videos stored in data
        center
71  constraint forall(req in REQUEST)(
72    let { int: rv = request[req, Rv];
73          int: re = request[req, Re];
74    } in ((videoSize[rv] > x \/ endpoint[re, K] = 0)
75    <-> vInDc[re, rv] = 1 )
76  );
77
78
79  % C2: Sum of all video sizes per endpoint <= cache's capacity
80  constraint forall(ca in CACHE)(
81      sum(vi in VID)(usedCache[ca, vi] * videoSize[vi])
82      <= x
83  );
84
85
86  % C3: compute the saving time per request
87  var int: savingTime = sum(req in REQUEST, ca in CACHE)(
88    let { int: rv = request[req, Rv]; % requested video
89          int: re = request[req, Re]; % requested endpoint
90          int: rn = request[req, Rn]; % number of requests
91          int: ld = endpoint[re, Ld]; % latency of data center to
              endpoint
92          int: lc = eConCache[re, ca];% latency of cache to endpoint
93    } in (
94        (ld - lc) *              % saving time when storing in cache ca
95        (1 - vInDc[re, rv]) *    % saving 0 ms if storing in data center
96        rn *                     % number of requests for video rv
97        usedCache[ca, rv]        % 1: is video rv is stored in cache ca;
98                                 % 0: otherwise
99  ));
100
101
102 % P2: pre-computation: total number of all requests
103 var int: nReq;
104 constraint nReq = sum(re in REQUEST)(request[ re, Rn ]);
105 % F1: returns 0 if the video is not stored in any other caches
106 %     except cach ca
107 function var int: selectedVideo(int: ca, int: rv) =
108     sum(c1 in CACHE where c1 != ca)(
109         bool2int(usedCache[c1, rv] = 1));
110
111 % F2: check capacity of cache ca if storing video vi
112 function var bool: hungryCache(int: ca, int: vi) =
113     x >=
```

```
114     (sum(vvi in VID)(videoSize[vvi] * usedCache[ca, vi])
115      + videoSize[vi]
116     );
117
118 % F3: check if cache ca is empty
119 function var bool: emptyCache(int: ca) =
120     sum(vi in VID)(usedCache[ca, vi]) = 0;
121
122 % C4: pre-computation predicted requests: unrequested videos
123 %     will be stored
124 constraint forall(en in ENDPOINT, ca in CACHE, vi in VID )(
125     eConCache[ en, ca ] > 0 /\ vInDc[ en, vi ] = 0 /\
126     emptyCache(ca) /\ hungryCache(ca, vi)
127     -> usedCache[ ca, vi ] = 1
128 );
129
130
131 % C5: store the requested videos iff there is still room in each
    cache
132 constraint forall(en in ENDPOINT, ca in CACHE, vi in VID)(
133     eConCache[ en, ca ] > 0 /\ vInDc[ en, vi ] = 0 /\
134     selectedVideo(ca, vi) = 0
135     -> usedCache[ ca, vi ] = 1
136 );
137
138 % C6: Avoid duplication
139 include "at_most.mzn";
140 constraint forall(vi in VID)(
141     at_most(1, [ usedCache[ ca, vi] | ca in CACHE ], 1)
142 );
143
144 % Decision variable: total time that saved for all requests
145 %     nReq is parameter, the division and multiplication
146 %     could be performed at the output phase.
147 % score = (savingTime / nReq ) * 1000
148 var int: score;
149 constraint score = savingTime;
150 solve maximize score;
151
152 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
153 % Output for test server
154 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
155 output [  %"V: " ++ show(v) ++ "\n" ++
156 %            "C: " ++ show(c) ++ "\n" ++
157 %            "E: " ++ show(e) ++ "\n" ++
158 %            "R: " ++ show(r) ++ "\n" ++
159           "S: \(score/nReq)" ++ "\n"];
160
161 % %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

there is a better way of handling them: take them out of the search space by real pre-computation!

implicative constraints are a source of inefficiency!

contradiction

of what?

* 1000

```
162 % % Output to file
163 % %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
164 % output ["\(n)"] ++
165 %        [ if j = 1 then "\n\(i-1) " % cacheId
166 %          else "" endif ++
167 %          if fix(usedCache[i, j]) > 0 /\
168 %             j >= 1 then "\(j-1) "  % videoId
169 %          else "" endif
170 %          | i in CACHE, j in VID] ++
171 %        ["\nscore: \((score/nReq)*1000)"
172 % %        ++"\n" ++ show2d(usedCache)
173 %        ];
```

B.1 Description

In the model, two 2D-matrix arrays `usedCache` and `vInDc` are defined. The first 2D-matrix array `usedCache[..,..]` represents decision variables of which videos will be put in which corresponding cache. The domain value of each element of `usedCache` is $\{0, 1\}$.

```
53 % ci : id of cache server being described
54 %      ( 0 .. C : 1.000 ), 0 : data center
55 % vi : id of videos stored in this cache server
56 %      ( 0 .. V : 10.000 )
57 array[CACHE, VID] of var {0,1} : usedCache;
```

In order to mark which videos are put in the data center because their sizes exceed the capacity of caches connecting to corresponding endpoint, another 2D-matrix array `vInDc[..,..]` is defined.

```
53 % ci : id of cache server being described
54 %      ( 0 .. C : 1.000 ), 0 : data center
55 % vi : id of videos stored in this cache server
56 %      ( 0 .. V : 10.000 )
57 array[CACHE, VID] of var {0,1} : usedCache;
```

There are 6 constraints, 3 functions, and 2 precomputations are introduced into this model. The decision variable `score` is bound to the `savingTime` to avoid the division / and *div*.

```
144 % Decision variable: total time that saved for all requests
145 %     nReq is parameter, the division and multiplication
146 %     could be performed at the output phase.
147 % score = (savingTime / nReq ) * 1000
148 var int: score;
```

The final score is computed in the output phase by dividing `savingTime` by total requests `nReq`, and then multiplying by 1000.

```
171 %        ["\nscore: \((score/nReq)*1000)"
```

Precomputation. The first precomputation (line 65-68) calculates the total number of used caches in the 2D-matrix `usedCache`.

While the second precomputation (line 102 - 103) iterates over all the requests and gives the total number of all requests.

Functions. Three functions are defined in this model. The first function `selectedVideo()` (line 105-109) takes two parameters, which are cache `ca` and video `rv`, and checks whether the video `rv` already stored in any other caches. It returns 0 if the video is not stored in caches other than cache `ca`.

The second function `hungryCache` (line 111-116) takes two parameters, cache `ca` and video `vi`. Before storing the new video `vi` into the cache `ca`, the spare capacity of the cache is checked to make sure that the total capacity does not exceed the given maximum capacity of the cache.

The last function `emptyCache` (line 118-120) takes one parameter cache `ca` and check whether the given cache `ca` is empty or not.

Model Constraints. The constraint C2 (line 79-83) guarantees that the total sizes of all stored videos in a cache does not exceed its maximum given capacity. The constraint C3 (line 86-99) computes the total number saving time of all caches and all requests. With all the empty cache, the unrequested videos will be stored in the cache. The constraint C4 (line 122-128). It iterates over the endpoint `ENDPONIT`, cache `CACHE`, and video `VID`. In this constraint, the unrequested video `vi` is stored in the cache `ca` under the following conditions: (1) there is connection between endpoint and cache `eConCache[ en, ca ] > 0`, (2) the considered video could be possible to store in the cache `vInDc[ en, vi ] = 0`, (3) the considered cache is empty `emptyCache(ca)`, (4) the cache doesn't exceed its limit when storing the video `hungryCache(ca, vi)`. The requested videos will be stored in the cache in constraint C5 (line 131-136) by using function `selectedVideo` to check whether there is connection between cache and endpoint `eConCache[ en, ca ] > 0`, the video `vi` is not stored in any other caches `selectedVideo(ca, vi) = 0`, and the size of videos does not exceed the capacity of the cache `vInDc[ en, vi ] = 0`. To avoid the duplication of stored videos, the constraint C6 (line 138-142) is defined. To all caches connecting to the endpoint, the `at_most()` restricts that each requested video could be stored only in one of those connected caches `at_most(1, [ usedCache[ ca, vi] | ca in CACHE ], 1)`.

Redundant Decision Variables and Channelling Constraints. Redundant decision variables `vInDc` are introduced into the model to mark which videos are stored in the data center. The `vInDc` reduces the search space when iterating over nested loops such as `VID`, `ENDPOINT`, and `CACHE`. The reified constraint in the model gives the solution of which videos are stored in the data center. When the size of a video exceeds the capacity of a cache or an endpoint does not have any connected cache server to store requested videos.

```
70 % C1: Channelling constraint: Mark which videos stored in data
     center
71 constraint forall(req in REQUEST)(
72   let { int: rv = request[req, Rv];
73         int: re = request[req, Re];
74   } in ((videoSize[rv] > x \/ endpoint[re, K] = 0)
75   <-> vInDc[re, rv] = 1 )
76 );
```

Implied Constraints. We do not see implied constraints could possibly be introduced into this model, ~~which thus also has no implied constraints.~~

(2)

Symmetry-Breaking Constraints. The 2D-matrix `usedCache[ CACHE, VID ]`, which represents the final result in the streaming videos problem, does *not* introduce the symmetries. Since each cache has different latency, swapping the cache rows in the `usedCache` might produce a non-optimal result. Similarly, swapping any number of columns which is corresponding to the stored videos in the `usedCache` solution might leads to a non-optimal result also.

Inference Annotations. We do not see inference annotations could possibly be introduced into this model, ~~which thus also has no inference annotations.~~

(?)

B.2 Implementation

The described model, with the prescribed comments, is uploaded as file `streamingVideos.mzn`.

Compilation and Running Instructions. In order to compile and run the model `streamingVideos.mzn` one must supply an input instance for all parameters such as `v, e, r, c, n, videoSize, endpoint, request`, and `eConCache` using the `-d` option in the command line. To output the result to a file, one must supply extra output file name using the `-o` option in the command line. There are 2 types of output in the model. One is for the test server which gives the output of the video size `V`, number of caches `C`, number of endpoints `E`, number of requests `R`, and total score `S`. Another one is for comparing with the expected output from Google requirement, which gives the total number of caches stored the videos, and the following lines describe which caches store which videos. All instances are put under the folder `streaming` when submitting the source code. Assuming that `minizinc` is one of the backends provided for the course, the model can be compiled and run for `warm_up.dzn` by typing `minizinc streamingVideos.mzn -d "streaming/warm_up.dzn"` at the command line. If one wants to output the result to a file, the complete command is `minizinc streamingVideos.mzn -d streaming/warm_up.dzn -o streaming/warm_up.out`

Sample Test-Run Command.

```
> minizinc streamingVideos.mzn -d streaming/warm_up.dzn
3
0 1 3
1 2
2 0
score: 562500.0
----------
==========
```

≠ Google's optimal?

solution: isn't that one

C Evaluation

We have chosen the backends for Gecode, Chuffed, Gurobi, OscaR.cbls, and Lingeling.

Table 3 gives the results for various instances 4 on the Streaming Videos model. The time-out was 600000 milliseconds. There is another evaluation of the alternative bin packing model, which is described in detail in D.3 with same time-out setting.

#=462.5 of Google's sub-optimum

Technology	CP		LCG		MIP		CBLS		SAT	
Solver	Gecode		Chuffed		Gurobi		OscaR.cbls		Lingeling	
Backend	Gecode		Chuffed		Gurobi		fzn-oscar-cbls		Picat-sat	
instance	score	time	score	time	score	time	score	time	score	time
warm_up$^{\psi}$	562.5	0.457	562.5	0.424	562.5	0.892	562.5	t/o	562.5	1.154
warm_up*	562.5	0.286	562.5	0.183	562.5	0.615	562.5	1.966	562.5	0.260
me_at_the_zoo$^{\psi}$	–	t/o	–	t/o	607.33	56.217	–	t/o	–	t/o
trending_today* $^{\psi}$	–	t/o	–	t/o	–	t/o	–	t/o	–	t/o
video_worth_spreading* $^{\psi}$	–	t/o	–	t/o	–	t/o	–	t/o	–	t/o
kittens* $^{\psi}$	ERR	–	ERR	–	ERR	–	ERR	–	ERR	–

Table 3: Results for our Streaming Videos model. (*) : *MiniZinc* 2.1.7, ($^{\psi}$) : *MiniZinc* 2.2.1

The experiment is done using two different version of *MiniZinc*, 2.1.7 and 2.2.1 as it is recently released. In the first experiment, all the instances are conducted using *MiniZinc* 2.1.7. The test results produced by *MiniZinc* 2.1.7, and *MiniZinc* 2.2.1 are marked by (*) and (ψ), respectively. In order to run the test in all backends, the final score computation is done at the output phase to avoid the division computations such as / and *div* which are not executable in *Chuffed* and *Gecode*. Ultimately, the significant difference between two version is the execution time which is illustrated in Figure 3. Overall assessment, *MiniZinc* 2.1.7 produces the result in the shorter time than the latest version 2.1.7. For instance, running *warm_up* instance, *MiniZinc* 2.2.1 produces the result in 0.457 second while *MiniZinc* 2.1.7 produces the result in 0.286 second, which means approximately 59% faster.

The model is tested using all five instances 4, with both *MiniZinc* 2.1.7 and *MiniZinc* 2.2.1. All the test results are shown in 3. To the instance *me_at_the_zoo*, the backend `Gurobi` is the best one among the others, since it could give the final score after 56.217 seconds while other backends timed-out. When testing with much bigger instances such as *trending_today*, and *video_worth_spreading*, all backends couldn't produce the final results after 600000 milliseconds. The instance *kittens* is the biggest and toughest instance that defeats all the backends, and ends up with the *ERR*.

Experiment with *MiniZinc* 2.1.7 The model has been tested with *5* instances 4. With the *warm_up* instance, our model gives the total score 562.5, which is better than the given score 462.5 in the Google specification. It's because the caches, which have the minimal latencies, are selected to store the requested videos in the first place. On the *warm_up* instance, we observe that all the chosen *Chuffed* backend wins overall with the execution time is 0.183 second, while the second rank is the *Picat-sat* backend with the execution time is 0.260 second. Other backends such as *Gecode*, *Gurobi*, and *fzn-oscar-cbls* give the results in 0.286, 0.615, and 1.966 seconds, respectively. In the experiment using older *MiniZinc* version, there is *no* time-out backend. In the next step, given a larger instance such as *me_at_the_zoo*, the winner solver is *Gurobi*, with the objective score is 56217. While all other solvers such as *Gecode*, *Chuffed*, *OscaR*, and *Lingeling* don't give any results and time-out. Starting from the medium instances such as *trending_today*, *video_worth_spreading*, and the largest *kittens* instance , backends are time-out and couldn't give any response with the time-out was 600000.

Experiment with *MiniZinc* 2.2.1 In the latest *MiniZinc* version, the winner backend is *Chuffed*, 0.424 second which is similarly to the older version of *MiniZinc*. The backend *Gecode*, which gives the result in 0.457 second, is the second rank. While it takes 0.892 second for *Gurobi* to produce the final score. The worst one is *Picat-Sat*, which gives the result after 1.154 seconds. The remarkable difference between the older and the latest *MiniZinc* version is that in the latest version, *fzn-oscar-cbls* is time-out while the result is given in the older version under the same time-out setting. Similar to the older version, when starting from the medium and the large instances, all the backends are time-out and couldn't give any results before timing-out.

Name	Videos	Cache Servers	Endpoints	Distinct Requests
warm_up	5	3	2	4
me_at_the_zoo	100	10	10	81
video_worth_spreading	10000	100	100	40317
trending_today	10000	100	100	95180
kittens	10000	500	1000	197987

Table 4: Instances of Streaming Videos model.

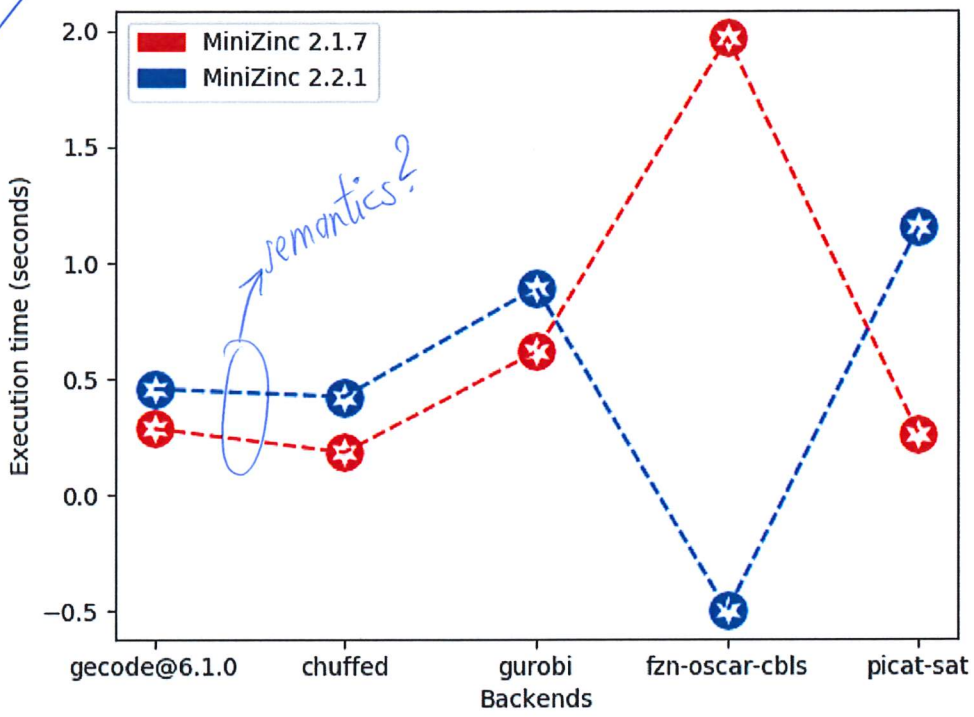

Figure 3: Comparison between *MiniZinc* 2.1.7 and *MiniZinc* 2.2.1 with *warm_up.mzn* instance

D Alternative Model

As the discussion in the presentation session, the `bin_packing` constraint could be used as an alternative model. The `bin_packing(int: c, array[int] of var int: bin, array[int] of int: w)` constraint requires that each item `i` be put into bin `bin[i]` such that the sum of the weights of each item, `w[i]`, in each bin does not exceed the capacity `c`. In this problem, with the view point of video serving network, capacity `c` corresponds to capacity `X` of each cache server. The weights of each item, `w[i]`, corresponds to the videos size `videoSize[i]`. Each *cache server* is corresponding to one *bin*, so `ca` cache servers corresponds to `ca` bins.

D.1 Description

Modelling the Streaming Videos problem with the new *bin_packing* view point, all the main constraints, functions, and precomputations in the previous model are reused (line 1 - line 145). In addition, the *bin_packing* model includes more constraints that consider the *caches* as *bins*, with maximum capacity and loading capacity. The *bin_packing MiniZinc* model could be found in Listing 2.

Listing 2: A *bin_packing MiniZinc* model for the Streaming Videos problem

```
147 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
148 %% Bin_packing model
149 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
150 include "decreasing.mzn";
151
152 array[CACHE] of var 0..x: cache_loads; % cache_loads
153
154 % the total load of each cache
155 constraint forall(ca in CACHE)(
156   cache_loads[ca] = sum(vi in VID)(usedCache[ca, vi] *
      videoSize[vi])
157 );
158
159 % the total load in the bin cannot exceed cache capacity
160 constraint forall(ca in CACHE) (cache_loads[ca] <= x);
161
162 % symmetry breaking:
163 constraint symmetry_breaking_constraint(forall(ca in 1..c-1) (
164       % if bin_loads[ca+1] is > 0 then bin_loads[ca] must be > 0
165       (cache_loads[ca + 1] > 0 -> cache_loads[ca] > 0)
166       /\ % and should be filled in order of weight
167       (cache_loads[ca] >= cache_loads[ca + 1])
168 ));
169
170 % symmetry breaking: use domain consistency for this constraint
171 constraint symmetry_breaking_constraint(decreasing(cache_loads) ::
    domain);
172
173 % another symmetry breaking: first bin must be loaded
174 constraint symmetry_breaking_constraint(cache_loads[1] > 0);
175
176 % alternative solve statement
177 solve :: int_search(
178          [usedCache[ca,vi] | ca in CACHE, vi in VID]
179          ++ cache_loads,
180          input_order, % first_fail,
181          indomain_max,
182          complete)
183     maximize score;
184
185 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
186 % Output for test server
187 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
188 output [
189 %           "Score: " ++ show(score) ++ "\n" ++
190           "Cache loads: " ++ show(cache_loads) ++ "\n" ++
191           "Used Cache: \n" ++ show2d(usedCache) ++ "\n" ++
192           ""];
193
194 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
195 % Output to file
196 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
197 output ["Google Output: \n"] ++
198         ["\(n)"] ++
199         [ if j = 1 then "\n\(i-1) " % cacheId
200           else "" endif ++
201           if fix(usedCache[i, j]) > 0 /\
202              j >= 1 then "\(j-1) "  % videoId
203           else "" endif
204           | i in CACHE, j in VID] ++
205         ["\nscore: \((score/nReq)*1000)"
206 %         ++"\n" ++ show2d(usedCache)
207         ];
```

Symmetry-Breaking Constraints. In the model, three symmetry breaking constraints are defined to hopefully produce the results faster. The first symmetry breaking constraint includes two other constraints, which is to put the videos in the cache servers in the increasing order, and another one is to guarantee that the loads of precedent cache server is not smaller than the next cache server. This symmetry breaking constraint might lead to the non-optimal solution in case of the latencies of first coming cache servers are not minimal. However, this trade-off could prune the search space and it is possible to give the result a bit faster.

```
162 % symmetry breaking:
163 constraint symmetry_breaking_constraint(forall(ca in 1..c-1) (
164         % if bin_loads[ca+1] is > 0 then bin_loads[ca] must be > 0
165         (cache_loads[ca + 1] > 0 -> cache_loads[ca] > 0)
166         /\ % and should be filled in order of weight
167         (cache_loads[ca] >= cache_loads[ca + 1])
168 ));
```

The second symmetry breaking constraint, which requires the `cache_loads` array be sorted in the decreasing order, uses the constraint annotation `domain`. The domain propagation is supported by some solvers, and advises the solver how the constraint should be implemented.

```
170 % symmetry breaking: use domain consistency for this constraint
171 constraint symmetry_breaking_constraint(decreasing(cache_loads) ::
    domain);
```

The third symmetry breaking constraint is to require the first `cache_loads` is loaded in the first place. This symmetry breaking constraint again could give the non-optimal solution because with all connected cache servers, the first cache server, which is loaded in the first place, could have the non minimal latency. However, in the huge search space, going through the search space to give all possible solutions in the limited time-out is obtainable.

```
173 % another symmetry breaking: first bin must be loaded
174 constraint symmetry_breaking_constraint(cache_loads[1] > 0);
```

Search Annotations. In the alternative *bin_packing* model for Streaming Videos problem, the `::int_search` annotation is used to compute the final *score* with the an array variables, which concatenate the `usedCache` array and `cache_loads` array. The next argument `input_order` specifies that the variables are chosen in the order that appear. To those chosen variables, the assignment annotation `indomain_max` will assign the largest video size in the `usedCache` and `cache_load` domain. Ultimately, the strategy annotation `complete` is specified.

```
% alternative solve statement
solve :: int_search(
          [usedCache[ca,vi] | ca in CACHE, vi in VID]
          ++ cache_loads,
          input_order, % first_fail,
          indomain_max,
          complete)
      maximize score;
```

D.2 Implementation

In this model, the load of each cache is computed and guaranteed that its load can not exceed the cache capacity. Furthermore, three more symmetry breaking constraints are also added in the *bin_packing* model. Those symmetry breaking constraints are described in detailed at D.1. Ultimately, search strategy is included to find the maximal score. An array of decision variables `cache_loads` is declared to capture the capacity of the cache servers, and keep updating their capacities when the new video is added. `array[CACHE] of var 0..x: cache_loads;` The package `decreasing.mzn` is included in the model to break the symmetry by requiring that the arrays `cache_loads` is in decreasing order.

```
% the total load of each cache
constraint forall(ca in CACHE)(
  cache_loads[ca] = sum(vi in VID)(usedCache[ca, vi] *
    videoSize[vi])
);

% the total load in the bin cannot exceed cache capacity
constraint forall(ca in CACHE) (cache_loads[ca] <= x);
```

D.3 Bin Packing Model Evaluation

We have chosen the backends for Gecode, Chuffed, Gurobi, OscaR.cbls, and Lingeling. Table 5 gives the results for various instances 4 on the Streaming Videos model. The time-out was 600000 milliseconds. The alternative model *bin_packing* is also tested using all five instances 4 using *MiniZinc* version 2.2.1. All the test results are shown in 5. To the instance *me_at_the_zoo*, the backend `Gurobi` and *fzn-oscar-cbls* produce the *ERR* rather than time-out. It's unclear to me why the *ERR* message is given. Other backend such as *Gecode*, *Chuffed*, and *Picat-sat* keep running until timing-out without producing any score. Similarly to the test results 3 with the previous model, when testing with much bigger instances such as *trending_today*, and *video_worth_spreading*, all backends couldn't produce the final results after 600000 milliseconds. The instance *kittens* is the biggest and toughest instance that defeats all the backends, and ends up with the *ERR*.

Technology	CP		LCG		MIP		CBLS		SAT	
Solver	Gecode		Chuffed		Gurobi		OscaR.cbls		Lingeling	
Backend	Gecode		Chuffed		Gurobi		fzn-oscar-cbls		Picat-sat	
instance	score	time	score	time	score	time	score	time	score	time
warm_up	562.5	2.614	562.5	0.936	562.5	1.088	562.5	t/o	562.5	1.966
me_at_the_zoo	–	t/o	–	t/o	ERR	–	ERR	–	–	t/o
video_worth_spreading	–	t/o	–	t/o	–	t/o	–	t/o	–	t/o
trending_today	–	t/o	–	t/o	–	t/o	–	t/o	–	t/o
kittens	ERR	–	ERR	–	ERR	–	ERR	–	ERR	–

Table 5: Results for the alternative *bin packing* model with *MiniZinc* 2.2.1

462.5 published by Google for their sub-optimum

not when I try

compilation takes ~15 seconds: ?

↓

symptom that something is way too complicated in your model

Gurobi solves it to optimality in 55 seconds

there were version changes recently, so please make sure to run the latest versions for your next experiments

E Conclusion

In this project, the first model is built firstly on the *MiniZinc* 2.1.7, and the first intention is to run on that version only, and all the test results are conducted on that version. However, the latest version, *MiniZinc* 2.2.1 is released lately, so the model is tested again based on the latest version. To make use of the results on *MiniZinc* 2.1.7, the comparison is done to figure out the differences between *MiniZinc* 2.1.7 and *MiniZinc* 2.2.1. The differences are illustrated in ??. On the other hand, the disadvantage of those backends is the division computation such as and *div*, which can be avoided by putting the division computation in the output phase.

References

[1] Google. Streaming videos, 2017. Available from `https://hashcode.withgoogle.com/2017/tasks/hashcode2017_qualification_task.pdf`.

[2] Google. Streaming videos data, 2017. Available from `https://hashcode.withgoogle.com/2017/tasks/qualification_round_2017.in.zip`.