Paper2 = Goal_1DL451_work_at_weekend

Towards Efficient Solvers for Optimisation Problems

Huu-Phuc Vo
Dept. of Information Technology
Uppsala University
Uppsala, Sweden
Email: huu-phuc.vo@it.uu.se

***Abstract*—Constraint programming (CP) is pervasive and widely used to solve real-time problems which input data could be scaled up to the huge sizes, and the results are required to be given efficiently and dynamically. Many technologies such as *CP*, *hybrid technologies*, *mixed integer programming (MIP)*, *constraint-based local search (CBLS)*, *boolean satisfiability (SAT)* could have different solvers and backends to solve the real-time problems. *Streaming videos* problem is the problem that requires to decide which videos to put in which cache servers in order to minimise the waiting time for all requests with a description of cache servers, network endpoints and videos are given. In this paper, we model the *streaming videos* problem in two different ways. The first model is implemented using heuristics, and the *global constraints* are used in the second model. The experiments are benchmarked using *MiniZinc*, which is an open-source constraint modelling language that can be used to model constraint satisfaction and optimisation problems in the high-level, solver-independent way. The aim of the paper is to benchmark these technologies to evaluate the execution time and final scores of the two models using large instances of input data from Google Hash Code.**

***Index Terms*—optimsation, constraint programming, modelling**

I. 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 *remote data centers* locating in thousands of kilometers away, *cache servers* which store copies of popular videos, and *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]. *MiniZinc* [3] is a constraint-based modelling language for satisfaction and optimisation problems such as *Streaming videos* problem with independent solving technologies which supports for diverse technologies' solvers for instances CP, CBLS [4], MIP, SAT, and SAT modulo theories (SMT). In this paper, the *bin-packing* approach, which is modelled in modelling language *MiniZinc*, is used to solve the *Streaming videos* problem in two different ways: use the built-in global constraint `bin_packing_load()`, and model the problem using a heuristic.

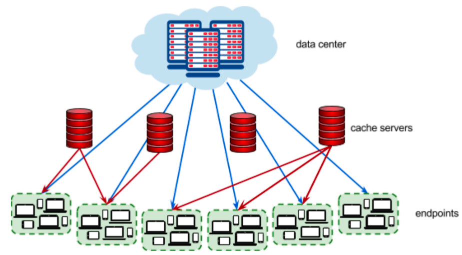

Figure 1: The video serving network

Our contributions are as follows

- The first contribution is to solve the *Streaming video problem* by modelling the bin packing algorithm using *MiniZinc* toolchain, which compiles the model into *FlatZinc*, which is the input of five different solvers.
- The second contribution is to perform experiments two approaches over five solvers inside *MiniZinc* using 2 versions of *MiniZinc*.

II. Backgrounds

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 words, the task is to *maximise* the average *saving time* for all given requests. Figure 1 illustrates the video serving network, which includes the data center, cache servers, and endpoints [1]. The *data center* stores *all videos*. The sizes of videos, 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 its maximum capacity. Every *endpoint* is connected to a 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.

Table I illustrates the input file [1]. 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. The conversion of the output instances which are compatible with *MiniZinc* is done by Python script together with precomputations [5], [6]. Table II illustrates the output file. Notice that the output example is not an 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. The paper makes the following contributions: microbenchmarks that compare the CP, LCG, MIP, CBLS, and SAT's `bin_packing_load` global constraint versus manual model.

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 I: Example of input file.

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 II: Example of output file.

III. Models

In this section, we present two approaches for the problem. In the first approach, the bin packing load algorithm is re-implemented in the manual model . The second approach uses the `bin_packing_load` constraint from *MiniZinc* library with search and strategy annotations to reduce the search space. The two approaches are evaluated on five solvers over two *MiniZinc* versions.

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

```
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 a corresponding endpoint, another 2D-matrix array `vInDc` is defined.

```
% 1: stored in data center, 0: not in dc
array[ENDPOINT, VID] of var {0,1}: vInDc;
```

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.

```
constraint score = savingTime;
```

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

```
output  ["\n score: \((score/nReq)*1000)"];
```

The first precomputation *P1* calculates the total number of used caches in the 2D-matrix `usedCache`. While the second precomputation *P2* iterates over all the requests and gives the total number of all requests. Three functions are defined in this model. The first function *F1* `selectedVideo()` 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 *F2* `hungryCache()` 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 *F3* `emptyCache()` takes one parameter cache `ca` and check whether the given cache `ca` is empty or not. The constraint *C2* guarantees that the total sizes of all stored videos in a cache does not exceed its maximum given capacity. The constraint *C3* computes the total number saving time of all caches and all requests. With all the empty cache, the unrequested videos are stored in the cache. The constraint *C4* iterates over the endpoint `ENDPOINT`, 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 does not exceed its limit when storing the video `hungryCache(ca,vi)`. The requested videos are stored in the cache in constraint *C5* 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* 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 `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.

```
constraint forall(req in REQUEST)(
  let { int: rv = request[req, Rv];
        int: re = request[req, Re];
  } in ((videoSize[rv] > x \/ endpoint[re, K] = 0)
  <-> vInDc[re, rv] = 1 )
);
```

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.

b) Global constraint model: The `bin_packing_load` constraint could be used as an alternative model. The `bin_packing_load(array[int] of var int: load, array[int] of var int: bin, array[int] of int: w)` constraint requires that each item `i` with weight `w[i]` be put into bin `bin[i]` such that the sum of the weights of items in each bin `b` is equal to `load[b]`. In this problem, with the view point of video serving network, capacity `load[i]` must be no greater than given capacity `X` of each cache server. The weights of each item, `w[i]`, corresponds to the videos size `reqVid[i]`. Each *cache server* is corresponding to one *bin*, so `C` cache servers corresponds to `C` bins. While the videos that are not requested or exceed the capacity of cache servers are stored in the data center. The *bin_packing_load* model includes constraints that consider the *caches* as *bins*, with maximum capacity and loading capacity. The videos that are stored in data center are implicitly captured by parameter `reqVid`. In the alternative *bin_packing_load* model for Streaming Videos problem, the `::int_search` annotation is used to compute the final *score* with array variables, which concatenate the `bin` array and `load` array. The next argument `first_fail` specifies that the variables are chosen in the order that appear. To those chosen variables, the assignment annotation `indomain_min` will assign the largest video size in the `bin` and `load` domain. Ultimately, the strategy annotation `complete` is specified.

```
solve :: int_search(
        bin ++ load,
        first_fail, indomain_min, complete)
        maximize savingTime;
```

IV. Experiments

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). The two models could be found at [7]. We have chosen the backends for Gecode, Chuffed, Gurobi, OscaR.cbls, and Lingeling. Table III gives the results for various instances IV on the Streaming Videos model. The time-out was 600000 milliseconds.

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 $(^{\psi})$, 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 2. 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 IV, with both *MiniZinc* 2.1.7 and *MiniZinc* 2.2.1. All the test results are shown in III. 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 could not 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*.

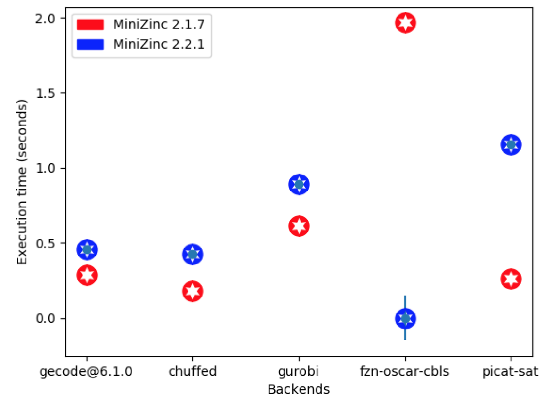

Figure 2: Comparison between *MiniZinc* 2.1.7 and *MiniZinc* 2.2.1 with *warm_up.mzn* instance

a) Experiment with MiniZinc 2.1.7: The model has been tested with *5* instances IV. 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* do not 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 could not give any response with the time-out was 600000.

b) 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

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	t/o	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$^{*\ \psi}$	–	t/o	–	t/o	–	t/o	–	t/o	–	t/o

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

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 could not 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 IV: Instances of Streaming Videos model.

V. Related work

MiniZinc is a standard modelling language for CP problems. The motto is *model once, solve anywhere*. Although *MiniZinc* may contain annotations to communicate with the underlying solver, its model is solver-independent. Most common global constraints, which defined over an arbitrary variables [8], and the separation between model and data are supported. It means a *MiniZinc* model can be instantiated by different data by defining as a generic template. *MiniZinc* supports sets, arrays, and user-defined predicates, overloading, and some automatic coercions. However, in order to easily map onto many solvers such as G12fd, lazyfd, and Chuffed, *MiniZinc* is still low-level enough. *MiniZinc* models are translated to *FlatZinc*, a low-level solver input language that is the target language for *MiniZinc*. When requiring by a CP solver, *FlatZinc* is translated easily into the required form. Since 2008, *MiniZinc* Challenge has been run every year to compare different solvers on the same benchmarks and to collect as well as develop new *MiniZinc* benchmarks. *Streaming videos* is one of the problems at the qualification round in the *Hash Code* 2017 competition running by Google. The coding contest *Hash Code* is sponsored by Google LLC. The competition is for the programmers who are living in Europe, the Middle East, and Africa. Participants must compete in a group of two to four members [9]. The contest consists of two rounds: the online qualification round and the final round. The *Streaming Video* data consists of four data sets which are in plain text files.

To the Streaming Video problem, the existing works based on different approaches such as using dynamic programming to implement the solution in C++, Python, Java. However, to the best of our knowledge, none of the existing works models the streaming video problem using constraint programming language and MiniZinc toolchain. In addition, in this work, we perform the experiments over five different solvers and two versions of MiniZinc toolchain to evaluate the two approaches.

VI. Conclusion and Future Work

The unique challenges of cache placement for video streaming services compared to web caching are caching strategies. Since video objects usually have a high read-to-update ratio compared to typical wed objects, such as web pages. In case of a web cache and a video cache with the same storage space, the first can store a larger number of objects than the latter. Therefore, different caching strategies are used in caches for video distribution mechanisms.

In this project, 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. The real question here is how can the *MiniZinc* model be improved to instantiate and give the result for the biggest data instance, *kittens*, whose size is up to 5.4 MB in text format. The *Streaming video* problem could be modelled by other modelling language and benchmarked with the same data instances to compare the performance and the efficiency with *MiniZinc* model.

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.

[3] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. Minizinc: Towards a standard cp modelling language. In Christian Bessière, editor, *Principles and Practice of Constraint Programming – CP 2007*, pages 529–543, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[4] Pascal Van Hentenryck and Laurent Michel. *Constraint-Based Local Search*. The MIT Press, 2005.

[5] P.F. Dubois. Python: Batteries included. *Computing in Science & Engineering*, 9(May), 2007.

[6] K.J. Milmann and M. Avaizis. Scientific python. *Computing in Science & Engineering*, 11(March), 2011.

[7] Huu-Phuc Vo. Streaming video models, 2019. Available from https://github.com/PhucVH888/ccgrid2019.

[8] Nicolas Beldiceanu, Mats Carlsson, Sophie Demassey, and Thierry Petit. Global constraint catalogue: Past, present and future. *Constraints*, 12(1):21–62, Mar 2007.

[9] Google LLC. Coding competition terms and conditions, 2019. Available from https://codingcompetitions.withgoogle.com/hashcode/rulesandterms.