Towards Efficient Solvers for Optimisation Problems

Huu-Phuc Vo

EasyChair preprints are intended for rapid dissemination of research results and are integrated with the rest of EasyChair.

October 21, 2019
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 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 constraint programming, hybrid technologies, mixed integer programming, constraint-based local search, boolean satisfiability 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 will model the streaming videos problem in two different ways. The first model will be implemented using heuristics, and the global constraints will be used in the second model. The experiments will be benchmarked using MiniZinc, which is an open-source constraint modelling language that can be used to model constraint satisfaction and optimisation problems in high-level, solver-independent way. The aim of the paper is to benchmark those 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—optimisation, 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 located 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 constraint programming (CP), constraint-based local search (CBLS) [4], mixed-integer programming (MIP), boolean satisfiability (SAT), SAT modulo theories (SMT), and hybrids, such as CP with lazy clause generation (LCG) [5]. In this paper, the bin-packing approach, which is modelled in modelling language MiniZinc, will be 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 heuristic.

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 word, 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 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.

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 will be done by Python script together with precomputations [6].
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.

III. MODELS

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 will be put in which corresponding cache. The domain value of each element of usedCache is {0,1}.

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.

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

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.

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, and VIDEO. 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 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, ENDPONIT, 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.

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 lead 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 will be 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.

```mini
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 [1].

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 (**), 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 IV.

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 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.

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 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.

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

---

1[https://github.com/PhucVH888/streamingVideos](https://github.com/PhucVH888/streamingVideos)
Table III: Results for our Streaming Videos model. (*) : MiniZinc 2.1.7, (v) : MiniZinc 2.2.1

Table IV: Instances of Streaming Videos model.

V. RELATED WORK

MiniZinc is a standard modelling language for constraint programming (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 instantiatted 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 Hash Code is a coding contest 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 as a team [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.

VI. CONCLUSION AND FUTURE WORK

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