Project Report - CMPT 417

Luiz Fernando Peres de Oliveira - 301288301 - lperesde@sfu.ca

I. Introduction

This project aims to solve the **Pizza** problem given in the *LP/CP Programming Contest 2015*, where some students in the University College Cork want to make a large order of pizza for a party such that they use as many vouchers collected throughout the year as they can. The final objective is to use the vouchers so to obtain all ordered pizzas with the least possible cost.

Because we want to minimize the total possible cost for all ordered pizzas, the problem is then an optimization problem and not only satisfiability, since we start with a target cost K with the sum of prices of all pizzas with no vouchers used and try to minimize the total cost by checking if the constraints are satisfiable for smaller K'. K is initialized with the total sum of prices of all pizzas with no vouchers used because this is the maximum possible cost for this problem and, therefore, if we were to choose any K larger than that, K could be improved to at least as good as the total sum of the pizza prices. We repeat the process by looking for satisfiable instances of a K' that are smaller than our previous K and stop whenever we cannot improve our K', meaning that K' is the least possible value of K.

In this document, we first specify the problem by defining our vocabulary $\mathcal{L}$ of functions and constant symbols for an $\mathcal{L}$-structure $\mathcal{M}$. We will then show that, to solve this problem, we must find a vocabulary $\mathcal{L}'$ (with a $\mathcal{L}'$-structure $\mathcal{M}'$) that extends $\mathcal{L}$ such that when we find a satisfiable instance of M for all constraints in $\mathcal{L}'$, we find a satisfying instance for the initial problem, since $\mathcal{M}'$ and $\mathcal{M}$ share the same universe M. After specifying our vocabularies and constraint relations, we will show our tests instances of the problem that were run on **Minizinc** solver and have their performance empirically evaluated with two solutions for the given problem. Finally, we will have a short discussion about the process of solving the Pizza problem.

II. Specification

The pizza problem is as follows:

- The final goal is to obtain all ordered pizzas for the least possible cost.
- A voucher is a pair of numbers (a, b) where you pay for a pizzas and obtain b pizzas for free as long as each of the b pizzas cost no more than each of the a pizzas.
- A voucher does not need to be completely used and not all vouchers need to be used.

A vocabulary $\mathcal{L}$ then can be defined by the symbols $[price, buy, free, n, m, k]$ where n is the number of pizzas, m is the number of vouchers, k is the cost bound, $price : [n] \to \mathbb{N}$ is an unary function that maps the price of each one of the n pizzas, $buy : [m] \to \mathbb{N}$ is an unary function that maps the number of pizzas that must be bought for each one of the m vouchers and $free : [m] \to \mathbb{N}$ is an unary function that maps the number of pizzas that come for free when a voucher is used, for each one of the m vouchers. The universe M are all the numbers appearing in the structure. The **Minizinc** equivalent form of $\mathcal{L}$ is:

int: n;
array[1..n] **of int**: price;
int: m;
array[1..m] **of int**: buy;
array[1..m] **of int**: free;
*% **k** is defined along with constraint C_{10} below*

Our problem is to find an assignment of pizzas and vouchers that minimize the total cost k and for this reason we will create a vocabulary $\mathcal{L}'$ that extends $\mathcal{L}$, where $\mathcal{L}$ and $\mathcal{L}'$ share the same universe M. Let $\mathcal{L}'$ be defined by the symbols in $\mathcal{L}$ and the symbols $[Paid, Used, Justifies, UsedFor]$, where the symbol $Paid : [n] \to \{0, 1\}$ is an unary symbol representing the set of paid pizzas, $Used : [m] \to \{0, 1\}$ is an unary symbol representing the set of used vouchers, $Justifies : [m] \to [n] \to \{0, 1\}$ is a binary symbol representing the set of vouchers v that will be used by paying for pizzas p and $UsedFor : [m] \to [n] \to \{0, 1\}$ is a binary symbol representing the set of free pizzas p that were obtained by using vouchers v. The **Minizinc** equivalent form of $\mathcal{L}'$ is:

array[1..n] **of var bool**: Paid;
array[1..m] **of var bool**: Used;
array[1..m, 1..n] **of var bool**: Justifies;
array[1..m, 1..n] **of var bool**: UsedFor;

To find a solution to instance $\mathcal{L}$ we must solve for vocabulary $\mathcal{L}'$ and $\mathcal{L}'$-structure $\mathcal{M}'$ and satisfy the problem constraints in **Minizinc**:

- C_1. If we paid for a pizza p, then p cannot be in the set of free pizzas

  constraint forall (p **in** $1..n$)
  $(Paid[p] \leftrightarrow$ **not exists**$(v$ in $1..m)(UsedFor[v, p]))$;

- C_2. If voucher v is used, then v must get at least one free pizza p with it

  constraint forall (v **in** $1..m$)
  $(Used[v] \leftrightarrow$ **exists**$(p$ **in** $1..n)(UsedFor[v, p]))$;

- C_3. Any used voucher v must be justified by paying for exactly some pizzas p

  constraint forall (v **in** $1..m$)
  ($Used[v] \rightarrow$
  sum(p in $1..n$)($Justifies[v,p]$) $>= buy[v]$);

- C_4. The number of free pizzas cannot be greater than what is possible by using voucher v

  constraint forall (v **in** $1..m$)
  (**sum**(p in $1..n$)($UsedFor[v,p]$) $<= free[v]$);

- C_5. For every two pizzas p_1, p_2, if p_1 was a pizza we got for free with voucher v and p_2 is a pizza we paid with voucher v, then the price of p_1 must be less or equal the price of p_2

  constraint forall($p1$, $p2$ **in** $1..n$ **where** $p1$!= $p2$,
  c **in** $1..m$)
  $((UsedFor[c,p1] \wedge Justifies[c,p2]) \rightarrow$
  $price[p1] <= price[p2])$;

- C_6.Two vouchers v_1, v_2 cannot be justified by using the same paid pizza p

  constraint forall ($v1, v2$ **in** $1..m$ **where** $v1$!= $v2$,
  p **in** $1..n$)
  ($Justifies[v1,p] \rightarrow$ **not**($Justifies[v2,p]$));

- C_7. We pay for every pizza p used to justify the use of a voucher v

  constraint forall (p **in** $1..n$, v **in** $1..m$)
  ($Justifies[v,p] \rightarrow Paid[p]$);

- C_8 and C_9. The pairs in $Justifies$ and $UsedFor$ can only be consisting of a voucher v and a pizza p in the form (v, p)

  constraint forall (v **in** $1..m$, p **in** $1..n$)
  ($Justifies[v,p] \rightarrow$ (v **in** $1..m \wedge p$ **in** $1..n$));
  constraint forall (v **in** $1..m$, p **in** $1..n$)
  ($Used[v,p] \rightarrow$ (v **in** $1..m \wedge p$ **in** $1..n$));

- C_{10}. The total cost (k, the sum of all paid pizzas) must be less or equal the sum of all pizza prices

  int: $total$ = **sum**($price$);
  var int: k = (**sum**(p **in** $1..n$)($Paid[p] * price[p]$));
  constraint $k <= total$;

- And finally, we want to minimize the whole cost K

  solve minimize k;

III. Testing

We created initially three test instances (the ones given in the *LP/CP Programming Contest 2015*) so to test if our specification was correct as defined on the contest and added seven so to evaluate how efficient the specification solve different instances of the same problem. It was finally added a last test instance with high numbers m and n so to test how the specification reacts to large problem instances. Each data in the test assign values to the number n of pizzas, the number m of vouchers, the map $price$ of pizzas prices and the maps buy and $free$ of vouchers, therefore establishing the universe M (all the numbers appearing in the structure). The detailed list of test instances were as below:

- Test 1: given in *LP/CP Programming Contest 2015*

  $n = 4$;
  $price = [10, 5, 20, 15]$;
  $m = 2$;
  $buy = [1, 2]$;
  $free = [1, 1]$;

- Test 2: given in *LP/CP Programming Contest 2015*

  $n = 4$;
  $price = [10, 15, 20, 15]$;
  $m = 7$;
  $buy = [1, 2, 2, 8, 3, 1, 4]$;
  $free = [1, 1, 2, 9, 1, 0, 1]$;

- Test 3: given in *LP/CP Programming Contest 2015*

  $n = 10$;
  $price = [70, 10, 60, 60, 30, 100, 60, 40, 60, 20]$;
  $m = 4$;
  $buy = [1, 2, 1, 1]$;
  $free = [1, 1, 1, 0]$;

- Test 4: Testing instances when all pizzas have same price

  $n = 10$;
  $price = [100, 100, 100, 100, 100, 100, 100, 100, 100, 100]$;
  $m = 5$;
  $buy = [3, 1, 1, 2, 3]$;
  $free = [3, 1, 1, 1, 2]$;

- Test 5: Testing instances when all pizzas have prices mixed

  $n = 9$;
  $price = [7, 20, 80, 47, 54, 68, 46, 38, 7]$;
  $m = 4$;
  $buy = [3, 5, 1, 4]$;
  $free = [1, 6, 2, 1]$;

- Test 6: Testing when the $n = 1$

  $n = 1$;
  $price = [1]$;
  $m = 4$;
  $buy = [3, 4, 5, 10]$;
  $free = [10, 5, 20, 15]$;

- Test 7: Testing with non obvious choice of vouchers

$n = 5;$
$price = [6, 91, 45, 45, 30];$
$m = 10;$
$buy = [4, 1, 4, 2, 1, 3, 4, 4, 4, 2];$
$free = [3, 4, 4, 1, 4, 4, 2, 3, 2, 3];$

- Test 8: Testing with a simple case of a choice of a voucher
  $n = 2;$
  $price = [7, 8];$
  $m = 1;$
  $buy = [1];$
  $free = [1];$

- Test 9: Testing multiple choices of vouchers
  $n = 5;$
  $price = [100, 99, 25, 10, 1];$
  $m = 5;$
  $buy = [1, 2, 3, 4, 5];$
  $free = [1, 1, 1, 1, 1];$

- Test 10: Testing with pizzas prices that are always double of the price of the previous
  $n = 7;$
  $price = [1, 2, 4, 8, 16, 32, 64];$
  $m = 5;$
  $buy = [3, 4, 5, 3, 4];$
  $free = [1, 1, 1, 2, 2];$

- Test 11: Testing how the specification reacts to large instances (check file **data11.dzn** for all pizza prices.
  $n = 30;$
  $price = [100, 100, 30, 44, 8, 44, ..., 38, 48, 30, 20, 50];$
  $m = 20;$
  $buy = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 2, 5, 3, 6, 7, 4, 3, 9, 10];$
  $free = [10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 4, 3, 2, 2, 8, 4, 3, 6, 5];$

We initially used the instances Test 1, Test 2 and Test 3 so to guarantee that the specification was minimally correct before testing the other test instances Test 4 through Test 10. Test 11 was too large and did not terminate. The conclusion is that the specification was working correctly as per the table below:

Instance	Least cost (k)	Output cost (k)	Runtime
Test 1	35	35	149 *msec*
Test 2	35	35	153 *msec*
Test 3	340	340	176 *msec*
Test 4	500	500	863 *msec*
Test 5	225	225	170 *msec*
Test 6	1	1	144 *msec*
Test 7	91	91	171 *msec*
Test 8	8	8	138 *msec*
Test 9	135	135	155 *msec*
Test 10	115	115	162 *msec*
Test 11	unknown	unknown	infinite

Obs.: The table shows that when a problem instance is very large, the problem might take too much time to answer (it was waited 1 hour and 6 minutes). It was expected as the optimization for the SAT problem is NP-hard and therefore there will exist cases where it is impossible to obtain a fast answer even with the best SAT solvers, unless P=NP.

IV. Empirical Performance

For means of empirical performance of the project, we will try to answer the question #1: "Which is the best way to express a constraint? There is often more than one reasonable way to express a particular constraint on the solutions in a particular language, and sometimes these choices can have a very large effect on running time. You can compare performance with different versions of one constraint, or compare two very different specifications".

While writing the specifications for the Pizza problem on **Minizinc**, there was an impression that when we specify constraints that use **sets** rather than **arrays** for satisfiability, their array versions are faster than the ones that use sets for given problem instances. It also seemed that constraints were faster in linear form, where "linear" means that the problem is not specified nestedly such as **constraint forall**(p **in** $1..n$, c **in** $1..m$)(...). Before the evaluation of the two specifications, the linear form seemed to be optimized and perform faster than their nested equivalent form **constraint forall**(p **in** $1..n$)(**forall** c **in** $1..m$)(...)). Below we show that while the data structure we use on constraints, **sets** versus **arrays** in this case, change the running time of **Minizinc**, **linear** versus **nested** constraint forms don't seem to have any (or little) runtime differences (e.g linear form could be a syntactic sugar for its nested form).

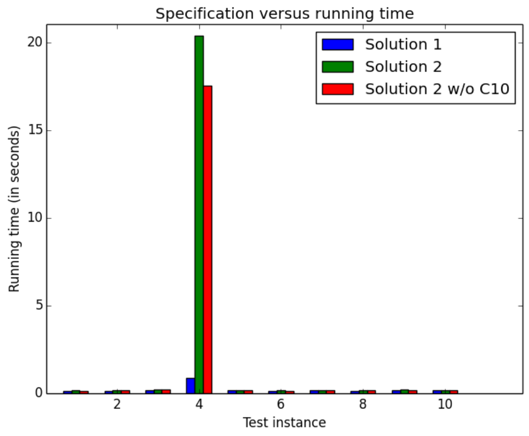
We created two different constraint specifications for the pizza problem and compared them. The first specification is the one specified in this document. The second specification is a modification of the first modification, where we use **sets** instead of **arrays** for the symbols $Paid$ and $Used$. Also, for the second specification we tested with and without constraint C_{10}, that seemed to have kept the correctness of the specification and reduced the total running time. Finally, we also only used linear constraint definitions on first specification and only nested constraint definitions on the second specification.

Spec 1:
array$[1..n]$ **of var bool**: Paid;
array$[1..m]$ **of var bool**: Used;

Spec 2:
var set of $1..n$: Paid; *// spec 2 redefines $Paid$*
var set of $1..m$: Used; *// spec 2 redefines $Used$*

As you will see on the table and graph below, this small change changes the running time greatly for some instances of the problem, such as **Test 4**.

Instance	Solution 1	Solution 2	Solution 2 w/o C_{10}
Test 1	149 *msec*	159 *msec*	152 *msec*
Test 2	153 *msec*	193 *msec*	154 *msec*
Test 3	176 *msec*	237 *msec*	218 *msec*
Test 4	863 *msec*	20*s* 419 *msec*	17*s* 548 *msec*
Test 5	170 *msec*	183 *msec*	190 *msec*
Test 6	144 *msec*	168 *msec*	128 *msec*
Test 7	171 *msec*	192 *msec*	169 *msec*
Test 8	138 *msec*	166 *msec*	164 *msec*
Test 9	155 *msec*	203 *msec*	155 *msec*
Test 10	162 *msec*	183 *msec*	164*msec*
Test 11	inf	inf	inf



The tests for empirical performance were performed using the solver *Gecode* 6.1.0 built-in on **Minizinc** and the evaluation environment is a Macbook Pro with a 2.2GHz quad-core Intel Core i7 processor and 16GB RAM.

With the tests performed above, we clearly see that constraints that use arrays rather than sets have a better performance for complex problems. It could be that sets are implemented as dynamic data structures and arrays are implemented as static data structures on Minizinc, or it could be that the operations for finding elements on sets (potentially $O(n)$, if well implemented $O(logn)$) are more expensive than finding elements on arrays $O(1)$ and that costed the total performance on our specifications. Sets however are better if compared memory-wise with arrays. Arrays always take the whole allocated memory to it and sets might request memory as it needs, consuming as much or less memory than arrays.

Two other interesting points that were found while evaluating our specifications are: First, linear iterators on constraints perform as well as nested iterators and might only be another way of doing the same; syntactic sugar. Second, constraints seem to have a fundamental role on the total runtime of the specification over a given solver. By removing a redundant constraint C_{10}, solution 2 running **Test 4** went from 20*s* 419*msec* to 17*s* 548*msec* in runtime, an improvement of 15% of the total runtime, what can be crucial in the process of solving hard problems.

Finally, Solution 1 outperforms Solution 2 with and without constraint C_{10} in all test instances (except Test instance 11, that never terminated for any of the three approaches).

V. Discussion

Solving problems with *SAT solvers* was initially very challenging because it is a completely different way of programming per se, as the problem is solved completely with a sort of declarative programming paradigm by constructing a specification with constraints and letting the computer do the actual problem solving, which is very different from the approach of solving problems I was accustomed (imperative programming). For this reason, I can say that I learned a new way of solving problems.

I also learned that there are Domain Specific Languages tools specialized in solving *SAT* problems and got intuition of how to make one with techniques such as breaking the problem in vocabularies and structures, then perform grounding in it so to later reduce the *SAT* problem to our problem and verify if the formula is satisfiable and that, by running *SAT* many times, you can optimize the solution for your problem.

Finally, I also learned that there many people in this field trying to improve the algorithms of *SAT solvers* and that depending on the tool you use and the constraints you specify, the running time of the process of problem solving might vary for better or for worse.

VI. Data

The files used in this project were:

- **pizza_solution.mzn**: The main specification file.
- **pizza_solution_comp.mzn**: The file with different implementation of constraints on **Minizinc** for comparizon.
- **pizza_solution_comp_without_c10.mzn**: Same as *pizza_solution_comp.mzn*, however we remove the constraint C_{10}.
- **comparizon.table**: Raw comparizon table.
- **data1.dzn**: Test 1 data file.
- **data2.dzn**: Test 2 data file.
- **data3.dzn**: Test 3 data file.
- **data4.dzn**: Test 4 data file.
- **data5.dzn**: Test 5 data file.
- **data6.dzn**: Test 6 data file.
- **data7.dzn**: Test 7 data file.
- **data8.dzn**: Test 8 data file.
- **data9.dzn**: Test 9 data file.
- **data10.dzn**: Test 10 data file.
- **data11.dzn**: Test 10 data file.