Project Report

CMPT 417

Shreya Jain (301391230)
Submitted To: David Mitchell
December 11, 2019

Table of Contents

Acknowledgements

This is to acknowledge Prof. David Mitchell for all his help and support given to me in order to complete this project. I also acknowledge my classmates of the course CMPT 417 for all the help they gave whenever I was stuck. Additionally, I acknowledge Sima, TA for the course CMPT 417, for the guidance she provided throughout the project.

Pizza Coupon Problem

The problem chosen for this project is the pizza coupon problem. The description of the problem is: There is a list of pizzas that need to be bought, each with a price. There is also a collection of coupons that are of the form “buy x pizzas, get y pizzas free”. Each of these coupons can be used to used to get a specified number of pizzas free, provided some other specified number of pizzas are paid for. Each pizza that is free by applying a coupon c must have a price no more than that of the cheapest paid-for pizza used to justify using coupon c. The objective is to find a selection of coupons, and “assignments” of pizzas to coupons (the paid pizzas and free pizzas associated with each coupon is used), that allows getting all the desired pizzas for total cost at most k. i.e. obtain all the ordered pizzas for the least possible cost. The actual cost is calculated by multiplying all the paid for pizzas by their price.

Using MiniZinc (the solver system used for the project), optimization of the problem was worked upon. Essentially, the total cost function of the pizzas was minimized to obtain the least possible cost to get all the desired pizzas. An assumption made here was that any pizza used to justify a coupon can be used only once.

Problem Specification

Let the instance vocabulary be [price, buy, free, n, m] where:

- n is number of pizzas and is a constant
- m is number of coupons and is a constant
- Unary function price: [n] → N, giving the price for each of the n pizzas
- Unary function buy: [m] → N, giving the number of paid pizzas required to justify using each of the m coupons;
- Unary function free: [m] → N, giving the number of free pizzas that can be obtained by using each of the m coupons;
- Cost bound k ∈ N

The following vocabulary symbols will also be further used:

- Unary relation symbol Paid, for the set of pizzas will be paid for
- Unary relation symbol Used, for the set of coupons that will be used
- Binary relation symbol Justifies, where Justifies(c, p) holds if pizza p is one of the pizzas we will pay for to justify using coupon c

- Binary relation symbol UsedFor, where UsedFor(c, p) holds if p is one of the pizzas we get free by using coupon c
- Φ is the formula for solving the problem

The constraints applicable are:

1. Pay for exactly the pizzas that are not obtained by applying the vouchers
2. Used is the set of applied vouchers
3. Any voucher that is applied must be Justified by sufficiently many purchased pizzas
4. The number of pizzas any voucher is applied to is not too large
5. Each free pizza costs at most as much as the cheapest pizza used to justify the application of the relevant voucher
6. Every pizza used to justify applying a voucher is paid for
7. The total cost is not too large (total of all pizzas)
8. Require that J(v,p) and A(v,p) hold only pairs consisting of a voucher and a pizza.
9. Any pizza used to justify a voucher can only be used once

The problem specification is then given by:

Given: A structure A for the vocabulary of the formula φ, and a valuation σ, $\#(\tilde{x}, \varphi(\bar{x}\bar{y}))\mathcal{A}[\sigma]$ is the number of distinct tuples $\bar{a} \in A^k$, where $k = |\bar{x}|$, such that $\mathcal{A} \models \varphi[\sigma(\bar{x} \to \bar{a})]$.

To find: A structure B that is an expansion of A to the vocabulary [price, buy, free, n, m, Paid, Used, Justifies, UsedFor] and that satisfies the following formula φ:

$\Phi = (\forall p[Paid(p) \leftrightarrow \neg \exists c\ UsedFor(c, p)]) \wedge (\forall c[Used(c) \leftrightarrow \exists p\ UsedFor(c, p)]) \wedge$

$(\forall c[Used(c) \to \#(p, Justifies(c, p)) \geq buy(c)]) \wedge (\forall c[\#(p, UsedFor(c, p)) \leq free(v)]) \wedge$

$(\forall c \forall p1 \forall p2[(UsedFor(c, p1) \wedge Justifies(c, p2)) \to price(p1) \leq price(p2)]) \wedge$

$(\forall p \forall c[Justifies(c, p) \to Paid(p)]) \wedge (sum(p, Paid(p), price(p)) \leq k) \wedge$

$(\forall c \forall p[Justifies(c, p) \to (c \in [m] \wedge p \in [n])]) \wedge (\forall c \forall p[UsedFor(c, p) \to (c \in [m] \wedge p \in [n])]) \wedge$

$(\forall c \forall p[Justifies(c, p) \to \neg \exists d\ (Justifies(d, p) \to (c\ !=\ d))])$

Code in MiniZinc for the above specification:

```
int: n; %number of pizzas
set of int: Pizza = 1 .. n;
array[Pizza] of int: price; %price is giving the price for each of the n pizzas

int: m; %number of pizza coupons
```

```
set of int: Coupon = 1 .. m;
array[Coupon] of int: buy; %buy is giving the number of paid pizzas required to justify using each of the m coupons
array[Coupon] of int: free; %free is giving the number of free pizzas that can be obtained by using each of the m coupons

array[Coupon, Pizza] of var bool: UsedFor; %UsedFor(c, p) hold if p is one of the pizzas received for free by using coupon c
array[Coupon, Pizza] of var bool: Justifies; %Justifies(c, p) hold if pizza p is one of the pizzas is paid for to justify using coupon c

var set of Pizza: Paid; %Set of pizzas will be paid for
var set of Coupon: Used; %Set of coupons that will be used

var int: cost = sum (i in Pizza) (bool2int(i in Paid) * price[i]);

%We pay for exactly the pizzas that we don't get free by using coupons:
constraint forall(p in Pizza)((p in Paid) <-> not exists (c in Used)(UsedFor[c, p]));

%Used is the set of coupons that is used:
constraint forall(c in Coupon)((c in Used) <-> exists (p in Pizza)(UsedFor[c, p]));

%Any coupon that is used must be justified by sufficiently many purchased pizzas:
constraint forall(c in Coupon)((c in Used) -> sum(p in Pizza)(bool2int (Justifies[c,p])) >= buy[c]);

%The number of pizzas any coupon is used for is not more than the number it allows us to get free:
constraint forall(c in Coupon)((c in Used) -> (sum(p in Pizza)(bool2int (UsedFor[c,p])) <= free[c]));

%Each free pizza costs at most as much as the cheapest pizza used to justify use of the relevant coupon:
constraint forall(c in Coupon) (
 forall (p,q in Pizza where p!=q) (
   ((UsedFor[c,p] /\ Justifies[c,q]) -> (price[p] <= price[q]))
  ));
```

```
%Every pizza used to justify use of a coupon is paid for:
constraint forall (p in Pizza) (
 forall (c in Coupon) (
   Justifies[c,p] -> (p in Paid)));

%The total cost is not too large:
constraint cost <= sum(p in Pizza)(price[p]);

%Justifies(c, p) and UsedFor(c, p) hold only of pairs consisting of a coupon and a pizza
constraint forall(c in Coupon) (
   forall(p in Pizza) (
     (Justifies[c,p]) -> (
        ((1 <= c ) /\ (c<=m)) /\ ((1 <= p) /\ (p<=n)))
   ));

constraint forall(c in Coupon) (
   forall(p in Pizza) (
     (UsedFor[c,p]) -> (
        ((1 <= c ) /\ (c<=m)) /\ ((1 <= p) /\ (p<=n)))
   ));

%Any pizza used to justify a coupon is used once
constraint forall(c in Coupon) (
 forall(p in Pizza)
   (Justifies[c,p] -> not exists(d in Coupon where c != d)(Justifies[d,p])
));

solve minimize cost;
output ["cost(" ++ show(cost) ++ ")\n"
   ];
```

Test Instances

A total of 10 test instances were constructed. 3 of them were taken from the official page of the LC/CP Programming Challenge to test the code with the given input and output samples. The remaining 7 of them were taken from the MiniZinc Challenge 2015 test instances. The test

instances taken from there were larger in size and hence were chosen so as to check the accuracy of the code and whether it will work for comparatively larger values/data. While running the program with the test instances, default settings in the Gecode 6.1.1 solver system were applied with only inputs differing every time.

Results

Input	**Output**	**Run Time**

n = 4;
price = [10,5,20,15];
m = 2;
buy = [1,2];
free = [1,1];
	cost(35)	218 msec

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];
	cost(35)	285 msec

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];
	cost(340)	411 msec

n = 20;
price =
[10,10,10,10,10,10,10,10,10,1
0,10,10,10,10,10,10,10,10,10,
10];
m = 5;
buy = [3,1,1,2,3];
free = [3,1,1,1,2];
	cost(120)	Stopped at 3min 11 sec

n = 18; price = [243,7031,4313,8033,6321,16 88,2132,4142,9593,5882,216 7,8072,4972,4594,1139,6071,	cost(29997)	5 min 42 sec

7764,500];
m = 4;
buy = [3,2,3,2];
free = [9,5,9,5];		

n = 19;
price =
[88,15,66,26,35,16,80,27,46,58,68,18,63,6,69,08,39,91,23];
m = 8;
buy = [8,8,7,8,7,7,7,8];
free = [7,7,0,7,0,0,0,7];
	cost(675)	Stopped at 9 min

n = 14;
price =
[87,74,63,34,63,16,97,75,08,86,17,61,31,04];
m = 4;
buy = [7,9,9,1];
free = [6,5,3,7];
	cost(207)	229 msec

n = 10;
price =
[50,60,90,70,80,100,20,30,40,10];
m = 9;
buy = [1,2,1,0,2,2,3,1,3];
free = [2,3,1,1,1,2,3,0,2];
	cost(210)	4 sec 315 msec

n = 20;
price =
[95,96,96,61,29,79,71,18,36,53,52,64,71,49,71,84,31,15,18,58];
m = 7;
buy = [5,5,5,5,5,5,5];
free = [9,9,9,9,9,9,9];
	cost(607)	Stopped at 10 min

n = 20;
price =
[70,10,60,60,30,100,60,40,60,20,80,70,90,56,90,87,50,50,20,80];
m = 8;
buy = [1,2,1,1,0,1,2,1];
	cost(747)	Stopped at 7 min

free = [1,1,1,0,1,1,0,1];		

Observations

The run time of the given instances set is highly dependent on the value of m and n. For smaller n and m i.e. number of pizzas and coupons respectively, the output is compiled in seconds. If the value of m is small, and even though n is comparatively bigger with large values of data, the output is obtained in competitive time. Similarly, if n is small with big value of m, it still gives an output in comparable time. If m and n are almost close to each other, the output is obtained in seconds. In all other cases, especially where the price set is full of large data values and buy and free sets are filled with data values having some kind of homogeneity, the solver takes a lot of time to provide an output.

Changes in the MiniZinc Code

To further investigate this observation, some changes were made in the MiniZinc code and specifications of the problem which were as follows:

```
int: n; % number of pizzas
set of int: Pizza = 1..n;
array[Pizza] of int: price;   % price of each pizza
int: m; % number of coupons
set of int: Coupon = 1..m;
array[Coupon] of int: buy;   % buy this many pizzas to use coupon
array[Coupon] of int: free;  % get this many pizzas free
set of int: UsedFor = -m .. m; % -i pizza is assigned to buy of coupon i
                    %  i pizza is assigned to free of coupon i
                    %  0 - no coupon used on pizza

array[Pizza] of var UsedFor: how;
array[Coupon] of var bool: used;

% assign right number of pizzas to buy order
constraint forall(c in Coupon)(used[c] <-> sum(p in Pizza)(how[p] = -c) >= buy[c]);
constraint forall(c in Coupon)(sum(p in Pizza)(how[p] = -c) <= used[c]*buy[c]);

% assign not too many pizzas to free order
constraint forall(c in Coupon)(sum(p in Pizza)(how[p] = c) <= used[c]*free[c]);
```

```
constraint forall(p1, p2 in Pizza)((how[p1] < how[p2] /\ how[p1] = -how[p2])
                        -> price[p2] <= price[p1]);

constraint forall(c1, c2 in Coupon where c1 < c2 /\ buy[c1] = buy[c2] /\ free[c1] = free[c2])
            (forall(p1, p2 in Pizza where price[p1] < price[p2])
                (how[p1] = -c2 -> how[p2] != -c1));

int: total = sum(price);
var 0..total: cost = sum(p in Pizza)((how[p] <= 0)*price[p]);

solve :: int_search(how, input_order, indomain_min, complete)
    minimize cost;

output
 ["\ncost("++show(cost)++");\n"];
```

In the above code, some of the previous constraints are removed or compressed and written as a single one. The binary relations are omitted completely and in place a new set with assumed property that -i corresponds to pizzas bought on coupon i while +i corresponds to free pizzas that are on coupon i. Also, the search annotation was used with the solve function to provide a specific way of how to search in order to find the solution to the problem. In this way, more details and constraints are provided to avoid a broader pool of possibilities. The test instances are kept intact to observe whether it's the problem instances that need to be worked upon or tweaking the specifications and code result in change of performance.

Experiment Details

In order to investigate, the previous test instances were tested with the modified specifications and code with the solver changed to Chuffed (0.10.4) as well. The runtime and the answers were compared with the previous results in order to understand what can be modified to get a better performance. The following results were obtained for the test instances:

Input	Output	Runtime
n = 4;	cost(35);	251 msec

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

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];
	cost(35);	265msec

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];
	cost(340);	273msec

n = 20;
price = [10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10] ;
m = 5;
buy = [3,1,1,2,3] ;
free = [3,1,1,1,2] ;
	cost(120);	>25min

n = 18;
price = [243,7031,4313,8033,6321,1688,2132,4142,9593,5882,2167,8072,4972,4594,1139,6071,7764,500] ;
m = 4;
buy = [3,2,3,2];
free = [9,5,9,5];
	cost(29997);	1 sec 96msec

n = 19;
price = [88,15,66,26,35,16,80,27,46,58,68,18,63,6,69,08,39,91,23] ;
m = 8;
buy = [8,8,7,8,7,7,7,8] ;
free = [7,7,0,7,0,0,0,7] ;
	cost(628);	1sec 649msec

n = 14;
price = [87,74,63,34,63,16,97,75,08,86,17,61,31,04];
m = 4;
buy = [7,9,9,1];
free = [6,5,3,7];
	cost(207);	329msec

n = 10;
price = [50,60,90,70,80,100,20,30,40,10] ;
m = 9;
buy = [1,2,1,0,2,2,3,1,3] ;
free = [2,3,1,1,1,2,3,0,2] ;
	cost(210);	436 msec

n = 20;
price = [95,96,96,61,29,79,71,18,36,53,52,64,71,49,71,84,31,15,18,58] ;
m = 7;
buy = [5,5,5,5,5,5,5];
free = [9,9,9,9,9,9,9];
	cost(582);	3sec 683 msec

n = 20;
price = [70,10,60,60,30,100,60,40,60,20,80,70,90,56,90,87,50,50,20,80];
m = 8;
buy = [1,2,1,1,0,1,2,1];
free = [1,1,1,0,1,1,0,1];
	cost(733);	> 20min

After comparing the above data with previous data, the following observations were made:

- For many test instances, by tweaking the specifications and code, it was possible to obtain accurate answers in comparable time. Hence, having alternative specifications improved the performance overall.
- The run time not really depend on the values of n or m; rather, they depend on the values present in the data set of the prices of pizza.
- Using Chuffed as a solver improved the overall performance of the solver system; but it was the change in the specifications that greatly affected the performance time.

Conclusion

Yes, the exploration of changes in specifications helped a great deal in identifying what exactly needs to be investigated in case of test instances. It is the set of values that need to be investigated rather the number of pizzas or coupons. Changes in specifications can drastically change the performance of a code and help in improving the run time. Changing the solver from default to Chuffed helps in improving the overall performance of the solver system.

From the above experiment, it is quite evident that there are a number of ways of writing the problem specifications which directly affect the performance of the problem solving system. In particular, it is important to be concise, efficient in declarative problem solving techniques as additional constraints and unorganized specifications lead to larger runtimes than necessary.

In future, I would work on how further can alternating the specifications can improve performance, Additionally, I would like to investigate how and why the nature of the data set values greatly affect the performance of the solver system and what could be done or modified with regards to the problem specification in order to incorporate all that data sets as well and get a solution to them in comparable time.

Appendix

List of files:

1. README file - Gives a list of files in the directory
2. Project Report - A report on the entire project, containing description of problem, problem specifications, results, investigations
3. Pizza Coupon MiniZinc Code - The code based on the initial given specifications
4. Modified Pizza Coupon MiniZinc Code - The code used for the modified constraints and problem specifications for investigation purposes
5. Test files - Containing 10 different test instances of the problem