CMPT417 Project Report - Pizza Problem

Erbo Shan 301243663 erbos@sfu.ca

1 Introduction

The pizza problem is originally from LP/CP Programming Contest 2015. this problem arises in the University College Cork student dorms. They are preparing for a big party, and they need to order large amount of pizza. A lot of student from that college have many of coupons so that they could get free pizza in purchasing pizzas.

For each coupon there is a constrain which requires student buy x pizzas in order to get y free pizza. A coupon is a pair of numbers, for example, $\langle 3, 1\rangle$, this indicates that students have to buy three pizzas and they are able to claim 1 free pizza, but the maximum cost of free pizzas cannot be greater than the cheapest pizza they pay for. Say students have coupon, $\langle 5, 3\rangle$, student could get three free pizzas with purchasing 5 pizzas.

$$\max_{cost(free(pizza))} < \min_{cost(buy(pizza))}$$

For this project, we need to minimize the total cost so that we could pay the least amount money for desired number of pizzas. In another word, we need to find optimized solution for this problem instead of finding a satisfiable answer. We could have a trivial total cost by adding up all pizzas' prices without applying any coupons, this value will be the maximum cost. Let k denote the maximum cost:

$$k = \sum_{i=1}^{n} cost(pizza[i])$$

We need to find the minimum cost with satisfying all the constraints, let *miniCost* denote the smaller cost, and we keep seeking for other satisfiable cost which has to be smaller than our previous *miniCost* then we overwrite our new *miniCost* to the old *miniCost*. We repeat this process til we cannot find such *miniCost* smaller than previous *miniCost*, which means we have the least possible value of *miniCost*. So the basic idea is keep try different combinations and find the such combination will lead to the least cost.

Ultimately, we will use **MiniZinc** solver to find the minimum cost.

2 Specifications, Constraints & Implementation

2.1 Specifications

Pizza Problem is defined as follow:

- We have a list of pizzas we could buy, and for each pizza there is a corresponding price.
- We have a list of coupons, $\langle x, y\rangle$, which means you need buy x pizzas and you will be eligible to get y free pizzas.
- For the each free pizza, the maximum cost of free pizzas cannot be greater than the minimum cost of the price of purchased pizzas.

We can define out instance vocabulary as [*price*, *buy*, *free*, *n*, *m*, *k*], *price*, *buy*, and *free* are unary function symbols and *n*, *m* and *k* are constant symbols.

- *n* is the number of pizzas we want to obtain.
- *m* is the number of coupons we have.
- *k* is the total cost without applying coupons, or we can say, *k* is the upper bound of total cost.
- *price* : [n] $\rightarrow \mathbb{N}$ is a unary function, it indicates the price of each pizza
- *buy* : [m] $\rightarrow \mathbb{N}$ is a unary function, it indicates the number of pizza you have to buy to claim free pizzas for coupon *c*.
- *free* : [m] $\rightarrow \mathbb{N}$ is a unary function, it indicates the number of pizza you can get for free for each coupon *c*.

We need to extend instance vocabulary by adding some relation symbols [*Paid*, *Used*, *Justifies*, *UsedFor*]

- *Paid* is a unary symbol indicates the set of pizzas we purchase.
- *Used* is a unary symbol indicates the set of coupons we use.
- *Justifies* is a binary symbol, *Justifies(c, p)*, if we use purchased pizza *p* to claim coupon *c* then *Justifies(c, p)* holds.

- *UsedFor* is a binary symbol, *UsedFor(c, p)*, if we use coupon *c* to get pizza *p* then *UsedFor(c, p)* holds.

2.2 Constraints & Implementation

There are 9 constraints which could lead to the minimized result, we will discuss those constraints later. Before we dive into implementations, let's get familar with the solver, **MiniZinc**. Basicly, MiniZinc is a constraint modeling language, it allows user specify constrained optimization and decision problem. The biggest advantage MiniZinc has is that it allows user to write models/constraints as writing First Order Logic/ Second Order Logic formulas. So you could implement and optimize problems in a high-level.

We will use following syntax *(some examples show the syntax of MiniZinc, we will go deep later)* for implementations of constraints:

- **int: n;** asks user to enter a value for n.
- **var 1..n: x;** the range of *x* is between 1 and n.
- **set of int: pizzas = 1..n;** defines a set of integer from 1 to n.
- **array[pizzas] of int: price;** creates an array of integer with length of pizzas.
- **array[coupons, pizzas] of var bool: UsedFor;** defines a binary relation *UsedFor* which indicates which pizza *p* is used for claiming free pizza by using coupon *c*.
- **array[coupons] of var bool: Used;** initializes an array of Boolean with length of coupons, and it indicates which coupon *c* is claimed.
- **forall, exists, not** are the keywords for $\forall$, $\exists$ and $\neg$.
- **/\\, \\/** are relation symbols for $\wedge$ and $\vee$, the syntax is compound of backslach and forceslash.
- **>, <** are relation symbols for greater and smaller.

2.2.1 Define functions and relations

Now, we have certain sense of syntax of MiniZinc, and we can dive into transforming Fist Order Logic formulas into code.

- First of all we need number of pizza and number of coupon:

```
int: n;
set of int: pizzas = 1..n;
int: m;
set of int: coupons = 1..m;
```

- After that we need to define unary functions:

```
array[pizzas] of int: price;
array[coupons] of int: buy;
array[coupons] of int: free;
```

- Some relation symbols will come in handy:

```
array[coupons, pizzas] of var bool:
                                Justifies;
array[coupons, pizzas] of var bool:
                                UsedFor;
array[pizzas] of var bool: Paid;
array[coupons] of var bool: Used;
```

2.2.2 Define Constraints

We need to define and use constraints to find a valid solution, if and only if solution satisfies all the constraints.

- Constraint 1: Set upper bound of minimum cost and make sure it will not be greater than the upper bound.

  minCost is the least cost to obtain all pizzas.
- Constraint 2: We only pay for the pizzas which is not the free pizzas by applyting coupon.
- Constraint 3: The *Used* set contains all the coupon we used.
- Constraint 4: We can only apply coupon *c* if there is sufficient number of pizza we purchase.
- Constraint 5: We cannot apply more than one coupon *c* to the pizza *p* which is justified for free pizza (*this one is missing in the lecture notes*).
- Constraint 6: For each coupon *c*, the number of free pizza we get cannot be greater than the number on coupon *c*.
- Constraint 7: The maximum price of free pizzas we claim cannot be greater than the minimum price of purchased pizzas.

- Constraint 8: For each pizza p, if we use p to justify/claim coupon$c$, p has to be paid, which mean p is in *Paid*.

- Constraint 9: We cannot justify/claim coupon c which is not in the coupon set.

2.3 Constraints Implementation

The following implementations are based on the constraints we define above.

```
var int: minCost = sum(p in pizzas
        where Paid[p] == true)(price[p]);
var int: maxCost = sum(price);

constraint forall(p in pizzas)
        (Paid[p] <-> not exists
        (c in coupons)(UsedFor[c,p]));

constraint forall(c in coupons)
        (Used[c] <-> exists
        (p in pizzas)(UsedFor[c,p]));

constraint forall(c in coupons)
        (Used[c] -> sum(p in pizzas )
        (Justifies[c,p] == true) >= buy[c]);

constraint forall(c in coupons)
        (Used[c] -> (sum(p in pizzas)
        (UsedFor[c,p]) <= free[c]));

constraint forall(p in pizzas)
        ((sum(c in coupons)
        (Justifies[c,p])<=1));

constraint forall(c in coupons)
        (forall (p1 in pizzas)
            (forall (p2 in pizzas)
            (( p1 != p2 /\ UsedFor[c,p1]
            /\ Justifies[c,p2])
            -> (price[p1] <= price[p2]))));

constraint forall (p in pizzas)
        (forall(c in coupons)
        (Justifies[c,p] -> Paid[p]));

constraint forall(c in coupons)
        (forall (p in pizzas)
        (Justifies[c, p] ->
        (c in coupons /\ p in pizzas)));
constraint forall(c in coupons)
        (forall (p in pizzas)
        (UsedFor[c,p] ->
        (c in coupons /\ p in pizzas)));
```

We need to find minimum cost:

```
solve minimize minCost;
```

3 Test Instances

In this section, we will use 12 test instances to test if our implementations are correct. The first three test cases are from the contest website, because there are correct result we could make sure our program is working properly. To run test case, you can simply run this command:

```
minizinc --solver Gecode
            cmpt417Pizza.mzn test*.dzn
```

If you did not include path of MiniZinc to your system. you shall run this command:

```
export PATH=/Applications/
    MiniZincIDE.app/Contents/Resources:$PATH
```

3.1 Test Instance 1-3:

Test 1 to 3 are from contest website which could help us make sure if our program generate result correctly.

```
# Test 1
n = 4;
price = [10,5,20,15];
m = 2;
buy = [1,2];
free = [1,1];
# Test 2
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
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 Result:

```
# Result of Test 1:
Minimum cost:
  35
Maximum cost:
  50

# Result of Test 2:
Minimum cost:
  35
Maximum cost:
  60

# Result of Test 3:
Minimum cost:
  340
Maximum cost:
  510
```

3.2 Test Instance 4:

There is no such a purchased pizza is used to claim more than 1 coupon:

```
n = 3;
price = [10, 10, 10];
m = 2;
buy = [1, 1];
free = [1, 1];
```

Test Result:

```
Minimum cost:
  20
Maximum cost:
  30
```

3.3 Test Instance 5:

The minimum cost of free pizza is no more than the cheapest purchased pizza:

```
n = 10;
price = [70,10,60,60,30,100,60,40,60,20];
m = 1;
buy = [1];
free = [10];
```

Test Result:

```
Minimum cost:
  100
Maximum cost:
  510
```

3.4 Test Instance 6:

The number of pizza we obtain is always equal the number of pizza we want to obtain. In a word, if the coupon c is buy one get ten free, but we want to have ten pizza, c cannot be used:

```
n = 10;
price = [100, 100, 100, 100, 100,
         100, 100, 100, 100, 100];
m = 1;
buy = [10];
free = [1];
```

Test Result:

```
Minimum cost:
  1000
Maximum cost:
  1000
```

3.5 Test Instance 7:

This test case is extended version of *Test Instance 6*, the most expensive free pizza cannot be greater than the least cost of purchased pizza. In test instance 7, we have 5 coupon, and each of them is buy one get one free:

```
n = 10;
price = [100, 90, 80, 70, 60,
         50, 40, 30, 20, 10];
m = 5;
buy = [1, 1, 1, 1, 1];
free = [1, 1, 1, 1, 1];
```

Test Result:

```
Minimum cost:
  300
Maximum cost:
  550
```

3.6 Test Instance 8:

If there is no coupon we can use, we have to buy all pizzas:

```
n = 18;
price = [1,4,3,2,4,1,3,4,1,5,
        3,4,3,4,6,5,6,3];
m = 0;
buy = [];
free = [];
```

Test Result:

```
Minimum cost:
  62
Maximum cost:
  62
```

3.7 Stress Testing:

We will do four stress testings to see how is its performance. This snippet of python code will help us generate test case:

```
# generate random integer values
from random import seed
from random import randint
seed(10)
    combination = [[15,30], [15, 30], [25, 30]]
    pizza = [], buy = [], free = []
    for i in combination:
        pizzas = i[0]
        coupon = i[1]
        for _ in range(pizzas):
            value = randint(10, 50)
            pizza.append(value)
        for _ in range(coupon):
            value = randint(10, 20)
            buy.append(value)
            value = randint(5, 10)
            free.append(value)
        pizza.clear()
        buy.clear()
        free.clear()
```

3.7.1 Stress Testing 1:

```
n = 15;
price = [27, 82, 18, 42, 25, 73,
        67, 70, 93, 58, 36, 22, 72, 13, 59];
m = 30;
buy = [23, 29, 10, 24, 18, 17,
        28, 13, 20, 10, 10, 10,
        30, 27, 10, 22, 16, 23,
        10, 26, 17, 24, 25, 27,
        17, 21, 17, 17, 24, 19];
free = [5, 8, 9, 10, 5, 6, 10,
        10, 7, 5, 10, 7, 10, 10,
        9, 8,9, 10, 6, 7, 7, 9,
        8, 9, 8, 9, 5, 8, 6, 10];
```

Test Result:

```
Minimum cost:
  652
Maximum cost:
  757
```

3.7.2 Stress Testing 2:

We are still using 15 as total pizza number, and 30 as coupon number. You may notice that, in 3.9.1, a lot of coupon are not valid, because we just want 15 pizzas, and more than 50 percent coupons require purchasing more than 15 pizza so that you could claim free pizza. So this test we need make sure each coupon is asking us to purchase less than 15, and see how its performance will be.

```
n = 15;
price = [46, 45, 39, 41, 24, 30,
        20, 49, 27, 40, 29, 29,
        42, 45, 43];
m = 30;
buy = [11, 14, 11, 9, 9, 9, 14,
       12, 12, 13, 10, 7, 15, 7,
       12, 12, 10, 8, 11, 14, 13,
       9, 13, 14, 10, 15, 11, 15,
       14, 14];
free = [5, 6, 8, 10, 8, 8, 6, 8,
        9, 9, 7, 9, 8, 8, 10, 6,
        9, 8, 5, 8, 5, 5, 8, 9,
        5, 5, 7, 5, 6, 5];
```

Test Result:

```
Minimum cost:
  311
Maximum cost:
  549
```

This test case takes more than 60 sec, we shall see the performance drops significantly, compared with test cases above

3.7.3 Stress Testing 3:

```
n = 25;
price = [48, 39, 41, 18, 39, 37,
        42, 38, 31, 26, 39, 37,
        31, 43, 43, 19, 25, 30,
        49, 12, 22, 47, 46, 50, 40];
m = 30;
buy = [15, 20, 18, 18, 12, 13,
       15, 19, 10, 20, 13, 20,
       19, 19, 20, 17, 10, 11,
       14, 16, 13, 14, 11, 10,
       19, 18, 11, 14, 12, 20];
free = [8, 6, 7, 9, 8, 6, 7,
        6, 5, 9, 6, 8, 6, 5,
        10, 7, 8, 10, 10, 9,
        10, 8, 8, 10, 10, 9,
        6, 7, 9, 7];
```

Test Result:

```
Minimum cost:
  559
Maximum cost:
  892
```

3.7.4 Stress Testing 4:

```
n = 25;
price = [38, 16, 36, 12, 10, 38,
         44, 14, 13, 32, 15, 19,
         16, 48, 39, 41, 18, 39,
         37, 42, 38, 31, 26, 39,
         37];
m = 30;
buy = [15, 20, 18, 18, 12,
       13, 15, 19, 10, 20,
       13, 20, 19, 19, 20,
       17, 10, 11, 14, 16,
       13, 14, 11, 10, 19,
       18, 11, 14, 12, 20];
free = [8, 6, 7, 9, 8, 6,
        7, 6, 5, 9, 6, 8,
        6, 5, 10, 7, 8, 10,
        10, 9, 10, 8, 8, 10,
        10, 9, 6, 7, 9, 7];
```

Test Result:

```
Minimum cost:
  470
Maximum cost:
  738
```

The last two test cases will not pass unless we use other solver rather than Gcode.

4 Benchmark

In this section, we will do some benchmarks based on different aspects, such as different solvers, different test cases and different numbers of threads.

4.1 Default settings

We will run all test cases on default settings, which means all solvers are using only **one** thread, and we will run each test cases **10** times. We will remove the highest and the lowest, and calculate the **mean**.

Test	Gecode6.11	Chuffed	COIN-BC
1	134	128	102
2	122	135	122
3	152	132	242
4	125	132	127
5	150	204	135
6	132	149	148
7	388	220	1765
8	152	149	103
9	807	372	1211
10	61100	473	1284
11	N/A	18350	120400
12	N/A	2692	60120

The unit is *ms*

4.2 Two Threads:

Since we cannot apply multi-threads on **Chuffed**, I jest leave it as it is in *Default Settings*, and we can remove **Chuffed** column from the table.

Test	Gecode6.11	COIN-BC
1	155	99
2	150	125
3	177	242
4	148	99
5	176	140
6	148	148
7	353	1793
8	152	103
9	582	1126
10	60800	1684
11	N/A	47713
12	N/A	27660

The unit is *ms*

As we can see, for test instance 9-12, there are slight improvement of performance. Let's see if increasing number of threads improves performance in complex test instances **9-12**.

4.3 Eight Threads:

Since we cannot apply multi-threads on **Chuffed**, I jest leave it as it is in *Default Settings.*

Test	Gecode6.11	COIN-BC
9	390	1148
10	23758	1695
11	75630	28225
12	N/A	42731

The unit is *ms*

As the table shown, multi-threading slightly boost the performance, but not too much. We can conclude that the performance of different solvers are vary. In complicated test instances, **COIN-BC** and **Chuffed** are outperformed **Gecode 6.11**, and **Chuffed** is better than **COIN-BC** in any circumstances

End of Part I

Part II

5 Analysis

We possibly notice that the performance of our solver goes downhill while we add more pizzas and more coupons. The outcome of *Test instance 9* is much more faster than *Test instance 10.* As we discussed in section 3.9.2, we have a lot of coupons which ask purchase more pizzas we want to obtain before we could claim free pizzas. For example, we want to obtain 15 pizzas, and all of the coupons are asking us to purchase **m** pizzas, where **m** is larger than 15, so that we could claim **n** free pizzas.

Let's introduce new vocabulary **valid***, if we say a coupon c* $\langle x, y\rangle$ *is valid or Valid(c), if and only if the x is less than the total number of piazzas we want to obtain.*

In test instance 9, solver would skip justifying all the coupons, because there is no pizza left to us to get for free or we have already obtain sufficient amount pizzas. That's the reason we could get the result within 10 seconds using *Gcode*, even though the size of test instance 10 is exact same as the size of test instance 9. So we need to run more test instances to check if this is the case, which is adding more *valid* coupons would slow the run-time.

5.1 More Test Instances

In this section, we will run more test case to see if *valid* coupons would influence the performance, or it is the reason slow the solver. We may need to refactor the format of output to visualize the result, which coupon *c* is used on which pizza *p*, and which pizza *p1* is used to claim coupon *c1.*

```
output[
        "Coupons: \n"
      ];
output[
        "  \(i). Buy \(buy[i]) get
        \(free[i]) free \n"
        | i in coupons
      ];

output  [
        "Pizzas Paid :\n  \(Paid)\n
         Coupons Used:\n  \(Used)\n
         Minimum cost:\n  \(minCost)\n
         Maximum cost:\n  \(maxCost)\n"
        ];
output[
      if fix(UsedFor[i, j])
      then
      "Applied coupon c\(i)
                  on pizza p\(j)\n"
      else ""
    endif  | i in coupons, j in pizzas
    ];
```

Sample output:

```
Coupons:
  1. Buy 1 get 1 free
  2. Buy 2 get 1 free
Pizzas Paid :
  [true, true, true, false]
Coupons Used:
  [true, false]
Minimum cost:
  35
Maximum cost:
  50
Applied coupon c1 on pizza p4
......
```

5.1.1 All coupon are not Valid

In test instance 9-12, there are only 15-25 coupons and we want to obtain 25 pizzas. For getting more accurate and precise comparison we will increase these numbers. We will test $< pizzas,\ coupons >$ as following combination:

- 25 pizzas, 35 invalid coupons
- 30 pizzas, 40 invalid coupons
- 35 pizzas, 45 invalid coupons
- 40 pizzas, 50 invalid coupons

If our assumption is correct, the running time will smaller than test instance 10-12, because none of these coupons will be justified and used. Again, each **invalid** coupon means it require m pizzas to claim free pizzas, where m is greater than the total number of pizzas we want to obtain.

As section 4 shows, *Gecode* is the slowest solver compared with *Chuffed* and *COIN-BC*, so we will benchmark all test instances based on *Gecode*. According to section 4, *Gecode* **cannot** even finish test instance 11 which has 25 pizzas and 30 coupons.

The reason we choose *Gecode* is that it is the slowest solver, and if it can finish in reasonable time, we can conclude that solver will skip justifying all invalid coupons. If that is the case, solver would not check if there is sufficient amount pizzas we can use for justifying coupon because we have already obtained all pizzas we want, and, furthermore *Gecode* could find solution instead of running forever.

Test	Coupons	Pizzas	Runtime(ms)
13	25	35	850
14	30	40	1225
15	35	45	1789
16	40	50	2354

Gcode 6.1

Some Test Results:

Output of our test results are too long, we pick two of them just for references.

```
Coupons:
  1. Buy 70 get 5 free
  2. Buy 63 get 5 free
  3. Buy 52 get 8 free
  4. Buy 69 get 7 free
  5. Buy 51 get 9 free
  6. Buy 58 get 10 free
  7. Buy 55 get 5 free
  8. Buy 57 get 9 free
  9. Buy 60 get 5 free
  10. Buy 68 get 6 free
  11. Buy 62 get 8 free
  12. Buy 58 get 9 free
  13. Buy 61 get 7 free
  14. Buy 68 get 9 free
  15. Buy 65 get 6 free
  16. Buy 64 get 5 free
  17. Buy 66 get 5 free
  18. Buy 53 get 8 free
  19. Buy 69 get 6 free
  20. Buy 61 get 6 free
  21. Buy 67 get 7 free
  22. Buy 56 get 8 free
  23. Buy 58 get 8 free
  24. Buy 64 get 8 free
  25. Buy 58 get 6 free
Pizzas Paid :
  [true, true, true, true, true,
  true, true, true, true, true,
  true, true, true, true, true,
  true, true, true, true, true,
  true, true, true, true, true,
  true, true, true, true, true,
  true, true, true, true, true]
Coupons Used:
  [false, false, false, false, false,
  false, false, false, false, false,
  false, false, false, false, false,
  false, false, false, false, false,
  false, false, false, false, false]
Minimum cost:
  1051
Maximum cost:
  1051
----------
==========
Coupons:
  1. Buy 66 get 8 free
  2. Buy 65 get 8 free
  3. Buy 63 get 9 free
  4. Buy 55 get 8 free
  5. Buy 64 get 7 free
  6. Buy 68 get 10 free
  7. Buy 69 get 7 free
  8. Buy 62 get 10 free
  9. Buy 68 get 8 free
  10. Buy 63 get 8 free
  11. Buy 66 get 7 free
  12. Buy 56 get 10 free
  13. Buy 68 get 9 free
  14. Buy 65 get 9 free
  15. Buy 54 get 6 free
  16. Buy 64 get 10 free
  17. Buy 52 get 6 free
  18. Buy 51 get 7 free
  19. Buy 65 get 9 free
  20. Buy 68 get 5 free
  21. Buy 53 get 10 free
  22. Buy 52 get 6 free
  23. Buy 62 get 10 free
  24. Buy 53 get 9 free
  25. Buy 55 get 9 free
  26. Buy 54 get 10 free
  27. Buy 70 get 8 free
  28. Buy 65 get 5 free
  29. Buy 66 get 5 free
  30. Buy 55 get 7 free
Pizzas Paid :
  [true, true, true, true, true,
  true, true, true, true, true,
  true, true, true, true, true,
  true, true, true, true, true,
  true, true, true, true, true,
  true, true, true, true, true,
  true, true, true, true, true,
  true, true, true, true, true]
Coupons Used:
  [false, false, false, false, false,
  false, false, false, false, false,
  false, false, false, false, false,
  false, false, false, false, false,
  false, false, false, false, false,
  false, false, false, false, false]
Minimum cost:
  1205
Maximum cost:
  1205
----------
==========
```

As we can see, none of those coupons are used for any pizzas we obtain. *Gecode* can even finish test instance with 3 seconds. If we look back at test instance 10, it takes almost 9 seconds to finish, and test instance only has 30 coupons, and 15 pizzas. Test instances has 40 coupons and 50 pizzas and it only takes 2 seconds.

5.1.2 All coupons are valid

We want to see if valid coupons will influence the performance by increasing the number of valid coupons. If this is the case, we could draw a conclusion, the complexity of coupons causes solver's performance, not only the size of test instance, the complexity of coupons also plays important role as well. So we will use same same test instance with more valid coupons. Section 4 shows that *Gecode* may not finish in reasonable time, we will use *Gecode* do the test instance first, if it cannot finish, then we will use other solver. We we should expect here is *Gecode* will not finshi single of them, because it cannot even pass test instance 11 which only has 15 coupons and 20 pizzas. In test instance 17-20 we have at least 25 coupons and 35 pizzas, but will run use *Gecode* first anyways. We will use following combination as test instance 17-20:

- 25 pizzas, 35 valid coupons

- 30 pizzas, 40 valid coupons
- 35 pizzas, 45 valid coupons
- 40 pizzas, 50 valid coupons

Test	Coupons	Pizzas	Runtime(ms)
17	25	35	NA
18	30	40	NA
19	35	45	NA
20	40	50	NA

Gcode 6.1

As our expectations, *Gecode* never finishes, so we will use *COIN-BC* and *Chuffed* to run test instances.

Test	Coupons	Pizzas	Runtime(ms)
17	25	35	2840
18	30	40	NA
19	35	45	NA
20	40	50	NA

Chuffed 0.10.4

Test	Coupons	Pizzas	Runtime(ms)
17	25	35	5663
18	30	40	39136
19	35	45	180500
20	40	50	NA

COIN-BC 2.10

Test Result:

```
Coupons:
 1. Buy 32 get 10 free
 2. Buy 22 get 6 free
 3. Buy 25 get 5 free
 4. Buy 37 get 5 free
 5. Buy 35 get 10 free
 6. Buy 38 get 8 free
 7. Buy 50 get 8 free
 8. Buy 28 get 5 free
 9. Buy 37 get 8 free
 10. Buy 22 get 10 free
 11. Buy 50 get 10 free
 12. Buy 40 get 5 free
 13. Buy 20 get 10 free
 14. Buy 38 get 9 free
 15. Buy 38 get 5 free
 16. Buy 48 get 7 free
 17. Buy 33 get 6 free
 18. Buy 37 get 8 free
 19. Buy 29 get 6 free
 20. Buy 36 get 7 free
 21. Buy 36 get 10 free
 22. Buy 37 get 9 free
 23. Buy 40 get 9 free
 24. Buy 49 get 8 free
 25. Buy 45 get 6 free
 26. Buy 22 get 5 free
 27. Buy 48 get 9 free
 28. Buy 40 get 9 free
 29. Buy 38 get 5 free
 30. Buy 37 get 9 free
 31. Buy 50 get 5 free
 32. Buy 29 get 10 free
 33. Buy 22 get 8 free
 34. Buy 43 get 8 free
 35. Buy 27 get 9 free
Pizzas Paid :
  [false,true,true,true,false,
  true,true,true,false,true,
  true,true,false,true,true,
  true,true,true,true,true,
  true,true,false,true,true,
  true,false,true,true,false,
  true,true,false,true,true,
  true,false,true,false,true,
  true,true,true,true,true]
Coupons Used:
  [false,false,false,false,false,
  false,false,false,false,false,
  false,false,true,false,false,
  false,false,false,false,false,
  false,false,false,false,false,
  false,false,false,false,false,
  false,false,false,false,false]
Minimum cost:
  1101
Maximum cost:
  1365
Applied coupon c13 on pizza p1
Applied coupon c13 on pizza p5
Applied coupon c13 on pizza p9
Applied coupon c13 on pizza p13
Applied coupon c13 on pizza p23
Applied coupon c13 on pizza p27
Applied coupon c13 on pizza p30
Applied coupon c13 on pizza p33
Applied coupon c13 on pizza p37
Applied coupon c13 on pizza p39
----------
==========
......
```

5.2 Observations

It is obvious that the complexity of coupons influences the performance a lot. By **complexity** of coupons, it means coupon c requires m purchased pizzas to claim free pizzas, where m is less than total amount of pizzas we want to obtain.

Gecode cannot handle complicated test instances well, as we can see, none of test instance in section 5.1.2 finishes while using *Gecode*. Even *COIN-BC* and *Chuffed* cannot finish all test instances. The performance for each different solver are vary. If *Chuffed* can finish certain test instance the running time will be the least, otherwise it never finishes. *COIN-BC* finished most of the test instances, but the performance is not good as *Chuffed*. If we look closely, we could notice that solver would generate all possible solutions and return the least cost solution.

So the solver is basically trying different combination of purchasing pizzas and justifying coupons based on our constraint and find the least cost. That is the reason why we increase the complexity of coupons, the performance goes downhill.

6 Assumptions & Validations

Since we have already figure out what influences performance in test instance level. We need to find if we can boost the performance on code we will come up some assumptions and test them to see whether we could get better performances.

6.0.1 Remove Unnecessary Constraints

As the section 2.2.2 shows, *We cannot justify/claim coupon c which is not in the coupon set*, and we have a implementation for this constraint. This constraints actually is not necessary because every coupon is in the list, and solver only choose coupon from the coupon list. So we could remove this.

After removing constraints:

Test	Coupons	Pizzas	Runtime(ms)
17	25	35	2906
18	30	40	NA
19	35	45	NA
20	40	50	NA

Chuffed 0.10.4

Test	Coupons	Pizzas	Runtime(ms)
17	25	35	4687
18	30	40	43887
19	35	45	180500
20	40	50	NA

COIN-BC 2.10

The performances do not change too much.

6.1 Assumptions

Is complexity of coupons the main factor slowing down the performance? The limitation of **MiniZinc** does not allow us to do more complicated test instances, we may need other tools to help us validate our assumption. Since we have already have contraints and we could implement this problem in C++, and then the C++ program will come in handy. First thing we need to figure out is that how to implement this in C++, and what is the best algorithm we should use for pizza problem.

6.1.1 Greedy Algorithm

Constraint 7: The maximum price of free pizzas weclaim cannot be greater than the minimum price ofpurchased pizzas. We could assume **Greedy Algorithm** will be an optimization for Pizza problem. Since the price of free pizzas we get cannot be greater than purchased pizzas. We could draw following formula:

$$\forall c \forall p_1 \forall p_2 ((p_1 \neq p_2 \land Justify(c, p_1) \land UsedFor(c, p_2) \rightarrow price(p_2) \leq price(p_1)) \quad (1)$$

We should maximum the sum of price of free pizzas, so that we could use less money to obtain more pizzas.

Let's say we have a coupon c, it requires purchasing m pizzas to get n free pizzas. To maximize the total sum of price of free pizzas we should buy top m the most expensive pizzas then we could claim top n the most expensive pizzas we do not purchase. For example, we have ten pizzas with prices:

```
price = [70,10,60,60,30,100,60,40,60,20];
buy = [3, 2];
free = [2, 3];
```

To get the least cost solution, we have to purchase the top 3 most expensive pizzas then get the top 2 most expensive pizzas for free.

```
purchase = [100, 70, 60]
getFree = [60, 60]
```

Then we need to purchase the top 2 the most expensive pizza we do not purchase yet, then claim 3 more free pizzas, and we will be done.

```
purchase = [60, 40]
getFree = [30, 20, 10]
```

So in total we spend 100 + 70 + 60 + 60 + 40 = 330. If we follow pattern we may boost the performance. But **MiniZinc** is a constraint programming language, feed in sorted list will not shorten the running time, and we are not allowed to write too many flow control in **MiniZinc**, so we will do a experiment in **C++**.

6.2 Validations

Let's start implementing this problem in **C++**, and see if our program could finish all test instances faster or this problem cannot be implemented in traditional programming language.

6.2.1 Brute Force & Greedy Algorithm

Inspired from the output from **MiniZinc**, we could force our program try different combinations and get the least cost. Since we will use **C++** to implement this, we could simply use *pair* in **C++** instead of using two list for initializing coupons. For example, we sort our price list first and coupon list, we go(buy) through the list, while we go through the list, we need to check whether we have sufficient amount pizzas we could use for getting free pizzas. If we have enough purchased pizzas, we claim coupon c and keep going through the list. Once we have the result, we done scanning the whole list and the result might be the correct result. Some pseudo-code may help you understand the idea:

```
price = {sorted price list};
coupons = {{a, b}, {c, d}, {e, f}...};
// {x, y} stands for buy x get y for free
// coupons is reversely sorted based on y
// {3, 6}, {5, 4}, {3, 3}.....
int purchased = 0;
int sum = 0;
int pizzasObtain = 0;
for(auto const& p: price){
  purchased++;
  pizzasObtain++;
  sum += p;
  if(coupons{x, y} where x == purchased){
      pizzasObtain += y + purchased;
      purchased = 0;
  }
}
```

We could run above program, and we could possibly get the least cost. We sort coupons list based on how many free pizzas we could get, and maximum the amount of free pizzas, so we could get the least cost. This approach looks intuitive and works fine, but we could notice that this implementation is not correct, yes, most of time using **Greedy Algorithm** is not correct. Let's say we have:

```
price = {70,10,60,60,30,100,60,40,60,20};
coupons = {{2, 5}, {1, 4}, {1, 4}};
```

Our algorithm will use coupon {*2 ,5*} first, so we need to purchase {*$100, $70*} pizzas and get {*$60, $60, $60, $60, $40*} pizzas for free, then purchase {*$30*} pizza and get {*$20, $10*} pizzas for free. In total, we spend 210. But the *CORRECT* solution is supposed to be **$160**, how so?

First of all, we purchase {*$100*} then use coupon {*1, 4*} to get four free pizzas, {*$70, $60, $60, $60*}. After that, we purchase {*$60*} pizza, and we could use another coupon {*1, 4*} and we could claim four free pizzas again, {*$40, $30, $20, $10*}. In total, we only spend 160, which is less than *210*.

The situation is pretty much same as **coin change** problem: *You are given coins of different denominations and a total amount of money amount. Write a function to compute the fewest number of coins that you need to make up that amount.*

Greedy algorithm does not work well in pizza problem. There might be other combinations which does not use the largest coupon but it will lead to the correct result. For this problem, we shall use BFS+Backtracking to find the optimized solution.

6.2.2 Breadth First Search & Backtracking

The idea of BFS and Backtracking approach is that we shall try all coupon and see if using certain coupon could lead us to a correct result. In a word we will find all possible solutions recursively. Before we dive into implementation, let's explain the idea.

- Every time we purchase a pizza, we need to check whether we obtain sufficient amount of pizzas so that we could use coupon. If we use certain coupon, we shall reset the amount of purchased pizza to zero, because we could not use two coupon on those purchased pizzas which we have already justified for free pizzas.

- Let's say we have coupons a, b, c, d, e...., we will test whether coupon a will lead to the least cost result, or coupon b will lead to the least cost etc. So we are not missing any combination of coupons.

- How to check if we obtain sufficient amount of purchased pizzas? We will use *unordered_map* to check if we have a such coupon we could apply. The time complexity of *map* is *O(1)*.

- If we obtain all pizzas we want, we need to update result. Is current result is the least cost, if so update it. If there is no coupon we could use, we need sum up the rest of pizzas which are not purchased yet. Let's say, we have obtained m pizzas, and we have n pizzas left, and there is such coupon c $\{x, y\}$ where x is less than n. We should pay the rest of pizzas.

- If we used all coupons, and there is still pizzas left, we shall purchase all of them, because there is free pizzas we could get. =

- We will mark coupon c if we use c, we do not want to use same coupon repeatedly. We will use

  ```
  vector<tuple<int, int, int>>
  ```

  for coupon, the first int stands for the number of sufficient pizzas, the second in stands for the number of free pizzas and the third int stands for if this coupon is used, *0* means not used, *1* means is used.

6.2.3 Implementations

List of source files we need and declare *namespace*:

```
#include <algorithm>
#include <climits>
#include <iostream>
#include <stdio.h>
#include <unordered_map>
#include <vector>
using namespace std;
```

We could define some types. Code will be more easier to read.

```
typedef vector<tuple<int, int, int>> Coupons;
typedef vector<int> Pizzas;
typedef unordered_map<int, Coupons> Map;
// define some types, easy to read
```

- For each coupon, we use $tuple < int, int, int >$ for coupon c.
- $get < 0 > (tuple)$ means how may pizzas we need to buy
- $get < 1 > (tuple)$ means how many pizzas we can get for free
- $get < 2 > (tuple)$ means if this coupon is used, 1: used, 0: not used

Define a *class* for our pizza problem, yes, OOP is important.

```
class PizzaProblem {
private:
    void find(Pizzas &pizzas,
        Coupons &coupons,
        Map &map, int start, int obtain,
        int cost, int total,
        int used, int &res){}
        // DFS function
public:
    int miniCost(Pizzas &pizzas,
        Coupons coupons){}
 }
```

start is the frist pizzas we shall purchase, **obtain** is the number of pizzas we obtain so far, **cost** is how much money we spend so far, we will use **cost** to update minimum cost, **total** is the number of pizzas we want to obtain, **used** is the number of coupon we use so far, and **res** is the solution.

- int miniCost()

```
int miniCost(Pizzas &pizzas,
                Coupons coupons) {
    Map map;
    int res = INT_MAX;
    auto cmp = [](tuple<int, int, int> a,
                tuple<int, int, int> b) {
      return get<1>(a) < get<1>(b);
    };
    // customized compare function
    sort(coupons.begin(), coupons.end(), cmp);
    for (auto const &c : coupons) {
      map[get<0>(c)].push_back(c);
    }
    // add coupon into map, use 'buy' as key,
    // 'free' as value, and the value is
    // vector<tuple>
    sort(pizzas.rbegin(), pizzas.rend());
    // sort pizzas array descending order
    find(pizzas, coupons, map, 0, 0, 0,
                pizzas.size(), 0, res);
    return res == INT_MAX ? -1 : res;
    // if res is equal to INT_MAX mean
    // there is no result, we return -1
};
```

First of all we have to sort our price vector, and the coupons vector. We need a lambda function to sort coupons vector because STL cannot sort vector of tuple types. We initilize res as *INT_MAX*, if we call *find()* and res is still *INT_MAX*, which means we could find any solution, otherwise return res.

- void find()

```
 void find(Pizzas &pizzas,
        Coupons &coupons,
        Map &map, int start,
        int obtain, int cost,
        int total, int used,
        int &res) {
    if (obtain >= total) {
      res = min(res, cost);
      return;
    }
```

If obtain is greater than total, which means we have obtained all pizzas we want, we need to update *res*, *res* will be updated if and only if we find new *res* which is smaller than previous one.

```
int left = total - obtain;
int canUseCoupon = false;
// if there is a such valid coupon we
// can use, if there is no coupon we
// could use, we should buy all
// left pizzas
for (int i = left; i >= 0; --i) {
  if (map.count(i) > 0) {
    int size  = map[i].size()
    for (int j = 0; j < size; ++j) {
      if (get<2>(map[i][j]) == 0) {
        canUseCoupon = true;
        break;
      }
    }
    if (canUseCoupon) {
      break;
    }
  }
}
```

If there is no coupons we could use, we have to buy the rest of pizzas. If there is coupons we could use, set *canUseCoupon* to true, which means we still have coupon to use.

```
// If we used all coupons or no valid
// coupons we need to buy
// the rest of pizzas
if (used == coupons.size()
        || !canUseCoupon) {
  int size = pizzas.size();
  for (int i = start; i < size; ++i) {
    cost += pizzas[i];
    // cout<<pizzas[i]<<endl;
  }
  res = min(res, cost);
  return;
}
```

Sum up all left pizzas and return.

If we still have valid coupon we could use, we will check if there is sufficient amount of pizza we purchase:

```
// We still have coupon we can use
if (canUseCoupon) {
    int purchased = 0;
    int sum = 0;
    for (int i = start; i < total; ++i) {
      sum += pizzas[i];
      if (map.count(++purchased) > 0) {
        // there is a coupon we could claim
        int size = map[purchased].size();
        for (int p = size - 1; p >= 0; --p) {
          if (get<2>(map[purchased][p])
                                   == 0) {
            // get<2>(map[purchased][p])
                                   == 0
            // means this coupon is not used
            // yet, we could try to use it;
            int freePizzas =
                  get<1>(map[purchased][p]);
            // how many pizza we can get
            // for free from this coupon;
            get<2>(map[purchased][p]) = 1;
            // set this coupon to 1, which
            // means this coupon is used;
            find(pizzas, coupons, map, i
                  + freePizzas + 1,
                 obtain + purchased
                 + freePizzas,
                 sum + cost, total,
                 used + 1, res);
            get<2>(map[purchased][p]) = 0;
            // set this coupon back to 0,
            // which means we no longer use it
          }
        }
      }
    }
  }
```

We use backtracking here, if we use coupon c, we need to mark it as used, so that we will never use c again, once we return from this recursive call, we need to set c back to not used. This is what backtracking means.

6.3 Test instance for C++ Program

Since we already have correct answers for test instances 1-20 using **MiniZinc**, we could use all of them to see if our C++ program work properly.

6.3.1 How to test

We are using test instance from **MiniZinc**, and we need to combine *buy* and *free* list into one vector, so this function will come in handy, it helps us get rid of redundant work, e.g manually merge two list into one vector.

```
Coupons convert(vector<int> buy,
                    vector<int> free) {
  // combine buy and free vectors
  // into one vector<tuple<int, int, int>>
  Coupons coupons;
  for (int i = 0; i < buy.size(); ++i) {
    coupons.push_back
            (make_tuple(buy[i], free[i], 0));
  }
  return coupons;
}
```

Initialize our first PizzaProblem object, this object provides us an API to run test instance.

```
int main() {
  auto pizzaProblem = new PizzaProblem();
  int res;
  Pizzas pizzas = {};
  Coupons coupons = {};
  vector<int> buy = {};
  vector<int> free = {};
```

Sample of input format:

```
  //     Web test 1
  pizzas = {70, 10, 60, 60, 30,
            100, 60, 40, 60, 20};
  buy = {1, 2, 1, 1};
  free = {1, 1, 1, 0};
  coupons = convert(buy, free);
  res = pizzaProblem->miniCost
                (pizzas, coupons);
  cout << "Correct answer: 340,
            our answer: " << res << endl;
```

Sample output:

```
Finished in 4 ms
Test 1 Correct answer: 0, our answer: 0
                               Time: 1.4e-05
Test 2 Correct answer: 46, our answer: 46
                               Time: 1.4e-05
......
```

Unit of **Time** *is second(s)*

6.3.2 Test Results

Now we need to run all test on C++ program, the results are as following:

```
Finished in 4 ms
Test 1 Correct answer: 0, our answer: 0
                              Time: 1.4e-05
Test 2 Correct answer: 46, our answer: 46
                              Time: 1.4e-05
Test 3 Correct answer: 340, our answer: 340
                              Time: 0.000223
Test 4 Correct answer: 35, our answer: 35
                              Time: 0.000217
Test 10 Correct answer: 311, our answer: 311
                              Time: 0.000365
Test 12 Correct answer: 545, our answer: 545
                              Time: 0.000233
Test 13 Correct answer: 1051, our answer: 1051
                              Time: 0.000218
Test 14 Correct answer: 1205, our answer: 1205
                              Time: 0.00023
Test 15 Correct answer: 1388, our answer: 1388
                              Time: 0.000285
Test 16 Correct answer: 1636, our answer: 1636
                              Time: 0.000345
Test 17 Correct answer: 856, our answer: 856
                              Time: 0.000172
Test 18 Correct answer: 952, our answer: 952
                              Time: 0.000344
Test 19 Correct answer: N/A, our answer: 1101
                              Time: 0.0002
Test 20 Correct answer: N/A, our answer: 1114
                              Time: 0.000225
```

Unit of **Time** *is second(s)*

Our program only takes 4ms to finish all test instance, which is 75000 times faster.

C++ program is way too faster than **MiniZinc**, let's try some crazy test instances, let's say 100 coupons (all valid coupons), 200 pizzas, and see how our C++ program performs.

```
Test 21 Correct answer: N/A, our answer: 2641
                              Time: 0.003194
```

It finishes within 4ms, this is pretty good performance. But we could notice that even though our coupons are valid, but all of them require purchasing at least 40 pizzas so that we could get free pizzas. Let's increase the complexity of coupons, so for each coupon it requires less purchased pizzas and we can get less free pizzas. This test instances may consume a lot of time, because the complexity of coupons is very high, and the number of desired pizzas is higher as well, let's see if our C++ program could finish in reasonble time? Still, we have 100 coupons and 200 pizzas.

Pizzas/coupons	Range(buy)	Range(free)	Time(ms)
200/100	80-100	40-60	0.94
200/100	60-80	40-60	6.15
200/100	40-60	20-40	16.8
200/100	20-40	20-40	397.1
200/100	10-20	10-20	NA

DFS/Backtracking

7 Conclusion & Disscussion

After running several massive test instances, we could draw a conclusion. The complexity of coupons slows down the performance of our program, not only **MiniZinc** but also **C++** program. It is very obvious that if we increase complexity of coupons, again increasing complexity of coupons means we could buy less pizzas to get less free pizzas, our program/solver will test more and more possible combination of coupons with purchasing different amount pizzas.

Even though our **C++** program runs 76000 times faster, it is still not able to finish the last test instance which has 100 coupons whose range of buy is from 10 to 20 and range of free is from 10 to 20, 200 pizzas need to obtain.

We could calculate the time complexity of our **C++** DFS+Backtracking algorithm. We denote m for the size of coupons, n for the size of pizzas in total. We have:

$$\begin{aligned} T(n) = m * T(n-1) + m * T(n-2) \\ + m * T(n-3) + ... + m * T(0) \\ = 2^n * m \end{aligned}$$

For the further study, we could figure out if there was a dynamic programming approach to solve this problem. If there was a such DP approach, we probably could have a implementation with $O(n^2)$ time complexity. Then we could discover more about pizza problem.

8 Appendix

8.1 Appendix A: Files

File	Description
cmpt417Pizza.mzn	Main **MiniZinc** program.
instanceGenerator.py	A **Python** script could generate test instances with different configurations.
PizzaProblem.cpp	A C++11 program which uses Depth First Search and Backtracking algorithm to solve pizza problem. There are 6 more complicated test instances inside of it.
README.md	A markdown file summarizes everything about this project.
test1.dzn	Test Instance.
test2.dzn	Test Instance.
test3.dzn	Test Instance.
test4.dzn	Test Instance.
test5.dzn	Test Instance.
test6.dzn	Test Instance.
test7.dzn	Test Instance.
test8.dzn	Test Instance.
test9.dzn	Test Instance.
test10.dzn	Test Instance.
test11.dzn	Test Instance.
test12.dzn	Test Instance.
test13.dzn	Test Instance.
test14.dzn	Test Instance.
test15.dzn	Test Instance.
test16.dzn	Test Instance.
test17.dzn	Test Instance.
test18.dzn	Test Instance.
test19.dzn	Test Instance.
test20.dzn	Test Instance.

8.2 Appendix B: Glossary

Term	Definition
valid	*A coupon c $\langle x, y\rangle$ is valid or Valid(c), if and only if the x is less than the total number of piazzas we want to obtain.*
Complexity of Coupons	*For each coupon c$\langle x, y\rangle$ x and y are relatively small, say if we want to obtain 100 pizzas, and each coupon only ask us to purchase 2-5 pizzas and we could get 1-4 free pizzas. We can say this sort of coupon has high complexity, if each coupon requires purchasing 80 pizzas and we could get 10 free pizzas, we will treat this coupon as low complexity*

References

[1] David Mitchell. *Notes on Satisfiability-Based Problem Solving Representing Problems: Examples from the 2015 LP/CP Programming Competition.* October 28, 2019