CMPT 417- Project Report

Rahul Anand – 301319328 – raa37@sfu.ca

PART 1

INTRODUCTION TO PROBLEM AND SOLVERS

The purpose for part 1 of the project was to study using declarative solving tools to solve instances of some application problem. The application problem that I choose to study is the *Pizza* problem as defined in the "LP/CP Programming Contest 2015" website. In part 1, I will be solving the *Pizza* problem as a decision problem meaning we should get either "satisfiable" or "unsatisfiable" as a result for the different test instances, and in part 2 we will look at the *Pizza* problem as a optimization problem.

The *Pizza* problem is about being able to purchase a specified set of pizzas, while using a specified set of coupons to minimize our total cost. However, this problem can prove to be quite difficult as there are restrictions on when we can use the coupons, what pizzas we use them on, and what pizzas we would like to get for free. In the decision problem version of this problem, we are given an upper bound "K" which will be the total amount of money we are willing to spend. Thus, if there is some combination of purchasing pizzas and using coupons that results in a cost less than or equal to "K", than our instance is satisfiable. However, if it is not possible to be able to achieve a cost less than "K", than our instance is unsatisfiable.

In Part 1, we will be using the MiniZinc solver, and the IDP solver. Both solvers were run via the command line, and not their respective IDEs. MiniZinc has different solver backends that it can use, such as Gecode, Chuffed, OR-Tools, etc. In part 1 of this report, we will be using MiniZinc with the Gecode solver backend.

PROBLEM SPECIFICATIONS

The *Pizza* problem is defined as follows (from LP/CP Programming Contest 2015).

The problem arises in the University College Cork student dorms. There is a large order of pizzas for a party, and many of the students have vouchers for acquiring discounts in purchasing pizzas. A voucher is a pair of numbers e.g. (2,4), which means if you pay for 2 pizzas then you can obtain for free up to 4 pizzas as long as they each cost no more than the cheapest of the 2 pizzas you paid for. Similarly a voucher (3,2) means that if you pay for 3 pizzas you can get up to 2 pizzas for free as long as they each cost no more than the cheapest of the 3 pizzas you paid for. The aim is to obtain all the ordered pizzas for the least possible cost. Note that not all vouchers need to be used, and a voucher does not need to be totally used.

Thus, based on the above definition, the basic objective and input for the pizza problem is described as:

- The final goal is to purchase all pizzas for the least amount of money
- A coupon is a pair of numbers (a,b) where "a" specifies how many pizzas we need to purchase in order to get "b" pizzas for free
- Not all coupons need to be used

A Pizza problem instance is then represented by

- **n**: The number of pizzas to obtain
- price: A list of n pizzas prices
- **m**: The number of coupons we have
- **free**: A list of size m that specifies how many free pizzas we get for using a certain coupon
- **buy**: A list of size m that specifies how many pizzas we need to purchase to use a coupon
- Therefore, if buy[2] = 3 and free[2] = 4, this means to use coupon 2, we first need to purchase 3 pizzas. Once we have purchased enough pizzas, we can then apply the coupon and get 4 pizzas for free.

The *Pizza* problem also introduces some restrictions, such as when you apply a coupon, the pizza that you get for free must be cheaper than all the pizzas you used to justify using that coupon. In addition, we cannot use one pizza to justify multiple coupons, and one coupon cannot be justified by a pizza that was bought to justify using a different coupon. Given all these constraints, we need to be able to formally represent them in MiniZinc and IDP. After doing some initial research, there are different ways in modelling the *Pizza* problem, however the way in which I chose to model the problem were based on the set of constraints given in class. Below is a list of the 8 constraints that I used to model this problem. Note, it would be unreadable if I were to include how I modelled these constraints in MiniZinc and IDP, which is why they are not show here. The constraints I used in MiniZinc and IDP are correctly labelled according to the following constraints, thus if you were to open pizza.mzn or pizza.idp, you can easily see the implementation of these rules.

1. We pay for exactly the pizzas that we don't get free by using coupons.
   $$\forall \mathrm{p}[\mathrm{Paid}(\mathrm{p}) \leftrightarrow \neg \exists \mathrm{UsedFor}(\mathrm{c}, \mathrm{p})]$$
2. Used is the set of coupons that we use.
   $$\forall \mathrm{c}[\mathrm{Used}(\mathrm{c}) \leftrightarrow \exists \mathrm{p} \mathrm{UsedFor}(\mathrm{c}, \mathrm{p})]$$
3. Any coupon that is used must be justified by sufficiently many purchased pizzas
   $$\forall \mathrm{c}[\mathrm{Used}(\mathrm{c}) \rightarrow \#(\mathrm{p}, \mathrm{Justifies}(\mathrm{c}, \mathrm{p})) \geq \mathrm{buy}(\mathrm{c})]$$
4. The number of pizzas any coupon is used for is not more than the number it allows us to get for free.
   $$\forall \mathrm{c}[\#(\mathrm{p}, \mathrm{UsedFor}(\mathrm{c}, \mathrm{p})) \leq \mathrm{free}(\mathrm{v})]$$
5. Each free pizza costs at most as much as the cheapest pizza used to justify use of the relevant coupon.
   $$\forall c \forall p1 \forall p2[(UsedFor(c, p1) \wedge Justifies(c, p2)) \rightarrow price(p1) \leq price(p2)]$$
6. We pay for every pizza used to justify use of a coupon.
   $$\forall \mathrm{p} \forall \mathrm{c}[\mathrm{Justifies}(\mathrm{c}, \mathrm{p}) \rightarrow \mathrm{Paid}(\mathrm{p})]$$
7. The total cost is not too large.
   $$\mathrm{sum}(\mathrm{p}, \mathrm{Paid}(\mathrm{p}), \mathrm{price}(\mathrm{p})) \leq \mathrm{k}$$
8. Justifies(c, p) and UsedFor(c, p) hold only of pairs consisting of a coupon and a pizza.
   1. $\forall \mathrm{c} \forall \mathrm{p}[\mathrm{Justifies}(\mathrm{c}, \mathrm{p}) \rightarrow (\mathrm{c} \in [\mathrm{m}] \wedge \mathrm{p} \in [\mathrm{n}])]$
   2. $\forall \mathrm{c} \forall \mathrm{p}[\mathrm{UsedFor}(\mathrm{c}, \mathrm{p}) \rightarrow (\mathrm{c} \in [\mathrm{m}] \wedge \mathrm{p} \in [\mathrm{n}])]$

In addition to these constraints, I added an additional 9th constraint that says, "No pizza is used to justify many coupons (e.g. 1 pizza cant be applied to multiple coupons)" Although this may not be necessary (as it was not mentioned in the class notes), it is important to enforce this rule.

SOLVER CORRECTNESS

Before creating test instances, I needed to verify that the solutions given by solvers were correct. I could not find any resources online of possible test instances that I can use, thus I had to rely on the few test instances that were provided in the “LP/CP Programming Contest 2015” website. I first verified the correctness of my solvers by inputting the test instances provided on the website, and ensuring that the output I received was the expected output. I then performed a series of other tests to ensure that I correctly implemented the constraints, which was done by modifying the given instances, and creating my own as well. Below is a list of some of the steps I performed to ensure solver correctness

- Modified the provided SAT instances to be UNSAT by changing the value of K (the cost bound) so that it is below the lowest possible cost, and ensuring that the solver produces UNSAT.
- Slightly changing values in the provided instances, manually checking by hand, and then ensuring the solver produces the same result.
- Creating brand new SAT and UNSAT instances, manually checking by hand, and then ensuring that the solver provides the correct output.
- Ensuring that the MiniZinc solver and IDP solver always give the same results (i.e. same min cost if instance is SAT, and output is UNSAT otherwise).

Most of the tests for correctness were done with reasonably sized problems similar to the one listed below

```
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]
```

Only a couple were done on large instances. This is due to the increasing complexity of such problems and I would no be able to verify by them by hand. An example of such a problem is the last test instance listed on the “LP/CP Programming Contest 2015” which I verified by matching the solution provided on the website, with the solution I was provided by my two solvers. Below is that instance.

```
n = 10
price=[70,10,60,60,30,100,60,40,60,
20]
m = 5
buy = [1,2,1,1]
free = [1,1,1,0]
```

The expected output listed on the website matched the expected output that I got from my 2 solvers, and after performing all the other tests, and having this larger test output correct results, I am confident that constraints are implemented correctly for the two solvers I am using (MiniZinc and IDP). In addition, while running my tests instances I always got the same results between MiniZinc and IDP, thus further confirming correct implementation of the constraints. In the next section I will talk about the tests instances I used to perform Part 1 of the project.

PROBLEM INSTANCES

I created 20 test instances to test the correctness of my solvers. 10 of these 20 instances are satisfiable, and 10 of the tests instances are unsatisfiable. The satisfiable instances include the 3 instances provided on the "LP/CP Programming Contest 2015" page, and the remaining 7 satisfiable instances were creating by modifying these 3 instances. I wanted to ensure I got a good range of test instances so for each size of n (i.e. the number of pizzas) I created two instances with different number of total coupons, values for the coupons, and of course different pizza prices. Thus, for my 10 satisfiable instances, I randomly generated 10 instances based on the initial 3 instances provided, and made sure that I had 2 randomly created instances for each 'n' I chose. The satisfiable instances are the first 10 instances provided, so for both MiniZinc and IDP, the instance files labelled 1 – 10 are satisfiable, and the remaining 11 – 20 are unsatisfiable.

I created the 10 unsatisfiable instances by also modifying the given instances provided on the "LP/CP Programming Contest 2015" page. There are two possibilities I considered as to why a Pizza problem instance may be unsatisfiable.

1. The minimal possible cost that the solver computes, is more than the value of "k" that was set (i.e. the total amount of money we want to spend)
2. We do not have enough money to buy all the pizzas.

To create UNSAT instances of the first type, I would first randomly create a new instance that is SAT by setting k to some large number so that a minimal solution is returned. I would then run that through the solver to get what the minimal possible cost is, then I would set k to be less than this computed minimal cost. Now I have generated a UNSAT instance, as the value of k is less than the minimal cost that is possible for that instance. To create UNSAT instances of the second type, I would simply make it that no coupon is ever used, buy setting the values in the "buy" list to large values, so that no coupon will ever get used. Since no coupon will ever get used, I would then set k to be less than the sum of the prices in the price list. Since no coupon will ever get used, we will need to buy all the pizzas without one, but since k is less than the total price of buying all the pizzas, the problem is now UNSAT.

For my UNSAT instances, instances 11 – 15 are UNSAT due to the first type of UNSAT instances I mentioned (the minimal possible cost the solver computes is more than the value of "k" that was set), instances 16 – 17 are UNSAT due to the second type of UNSAT instances I mentioned (we do not have enough money to buy all the pizzas), and lastly instances 18 – 20 are UNSAT due to the first type of UNSAT instances that were mentioned. I chose to have more UNSAT instances of the first type as I found it a more interesting problem for the solver to solve than that of the second type, where the solver will be able to quickly tell that a problem is UNSAT. However, both are valid test cases which is why I included them both. Below is a reference I used in creating the UNSAT instances.
https://piazza.com/class/kf5182t3yatmq?cid=138

SOLVER SETTINGS

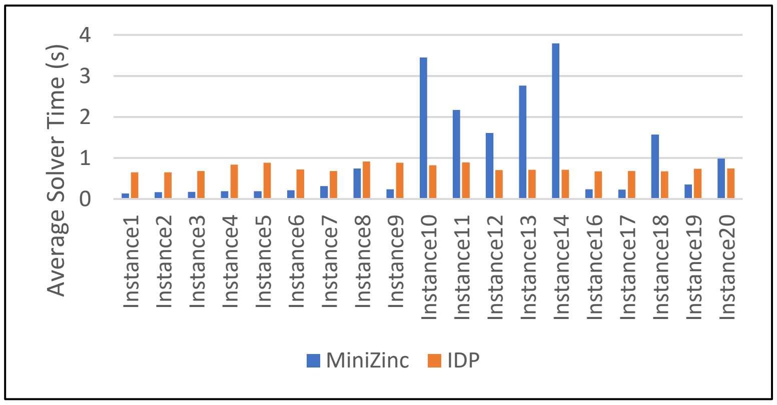
Both solvers were used with the default settings, and were executed via the Linux command line. MiniZinc was run with the Gecode backend solver, which is the default solver.

RESULTS

I ran each test instance 3 times and then calculated the average of the results. I chose to do this as solver time can vary a bit between runs, and thus decided that it would be a good idea to perform multiple runs to help average out any inconsistencies that may occur. Below is a table of the average solver time taken by MiniZinc and IDP for each problem instance.

	MiniZinc	**IDP**
Instance1	0.13	0.65
Instance2	0.17	0.65
Instance3	0.17	0.68
Instance4	0.19	0.84
Instance5	0.19	0.89
Instance6	0.21	0.72
Instance7	0.31	0.68
Instance8	0.75	0.92
Instance9	0.23	0.88
Instance10	3.45	0.82
Instance11	2.17	0.89
Instance12	1.61	0.71
Instance13	2.76	0.71
Instance14	3.79	0.71
Instance16	0.24	0.67
Instance17	0.23	0.68
Instance18	1.57	0.67
Instance19	0.35	0.74
Instance20	0.99	0.74

After running all 20 test instances through MiniZinc and IDP, the results are quite clear. The IDP System can come to a solution much quicker than MiniZinc can. For smaller solutions where MiniZinc was able to find a solution in less than a second, it seems like IDP would take a bit more time, such as around 100 to 400 more milliseconds, but for larger problems where MiniZinc took a couple seconds, or a couple of minutes, IDP was able to come to a conclusion on those tests in less than a second. Below is a visualization of the time each solver took for the test instances.



As you can see from the above graph, IDP usually took a bit more time for smaller instances where the solution can be found in less than a second, but for instances where the solution may take a second to a couple of seconds to solve with MiniZinc, IDP was much faster at being able to come to a conclusion. You may have noticed in the graph that Instance 15 is missing. This is because Instance 15 had a much larger run time compared to the other instances, and thus I took it out from this graph so that we can more easily see and compare the other instances with each other.

Below is the MiniZinc representation of Instance 15 (IDP not shown as the syntax for IDP is more complicated, but it still follows the below test case)

```
n=14
price=[3,6,33,1,4,2,7,4,77,1,55,12,12
,43]
m = 7
buy = [8,6,4,7,9,5,44]
free = [1,6,3,2,5,7,3]
k=221
```

Using MiniZinc, this instance took 196.552 seconds (3 minutes and 16.552 seconds). Using IDP, this instance took only 0.63 seconds. This is a huge gap between the two solvers performance, and thus shows how much faster IDP can be in comparison to MiniZinc. Thus, based on the benchmarks I have collected, it looks to be the case that IDP is faster than MiniZinc when it comes to larger and more complicated instances of the Pizza problem.

The benchmark times that I collected for the two solvers can be found in the excel file labelled “Data”.

PART 2

INTRODUCTION TO PROBLEM AND SOLVER

In part 1 of the project, I looked at how we can model and solve the *Pizza* problem using the MiniZinc and IDP declarative problem solving tools. I discovered that for smaller test instances, MiniZinc would reach a conclusion about satisfiability a bit quicker than IDP (e.g., MiniZinc taking 0.16s and IDP taking 0.64s for a small test instance). However, for the larger and more complicated test instances, where we had many pizzas and coupons, IDP would outperform MiniZinc every time, and often would take less than half the time that MiniZinc would take.

Based on these observations it appears that IDP should be the tool of choice when you are working with harder and larger problems. However, I found working with MiniZinc and its abundance of online documentation and resources to be much more detailed and helpful than what is provided for the IDP system. In addition, getting started with MiniZinc was much easier than with IDP, as the installation process is simpler, and the syntax of MiniZinc specifications are more intuitive and easier to understand than that of IDP.

This tradeoff of performance and simplicity between the chosen solving tools is what motivated my work for part 2 of the project. For part 2 of the project, I aimed to improve the solver performance of MiniZinc, so that it matches and/or outperforms the performance that IDP exhibits. There are two main questions I asked when trying to improve MiniZincs performance.

1. Can we modify and/or add redundant clauses to our specification to improve solver performance?
2. Can we adjust and or modify the parameters that MiniZinc utilizes to improve solver performance?

I will create a new set of test instances that are different than those from part 1, to evaluate the solvers performance in regard to these 2 questions. Thus, I will have be benchmarking the time taken by three different versions of the MiniZinc solver against this new set of test instances. The three different versions of the solver will be

1. A Unmodified MiniZinc specification, with default settings and the original set of constraints defined in part 1. This version of the solver will serve as the base to which we compare the other versions to.
2. A modified MiniZinc specification where the constraints are modified. This version of the solver will try to answer Question 1 above.
3. A unmodified MiniZinc specification with the original set of constraints defined in part 1, but the solver will be run with some non-default parameters. This version of the solver will try to answer Question 2 above.

In part 1 of the project, I looked at the *Pizza* problem as a decision problem, where we want to check if we can find some combination of buying pizzas and using coupons such that our total cost is less than some value K. If we can get our total cost less than K, the problem is satisfiable, if not, the problem is unsatisfiable. In part 2, I will look at the *Pizza* problem as an optimization problem. This means that there is no K value that serves as an upper bound on cost. We are simply trying to find the lowest possible cost for our given problem instance.

PROBLEM INSTANCES

In part 1 of the project, I tested the solvers correctness with 20 test instances, 10 being satisfiable, and 10 being unsatisfiable. In part 2 of the project, I created 15 new instances, and since I am doing the optimization version of the *Pizza* problem, a minimum cost will always be returned for each instance.

I chose to create 15 new instances rather than re-use the 20 test instances created in part 1, as many of the instances in part 1 were testing different edge cases, using smaller sizes of N so that we can easily verify solutions, making sure that we have both SAT and UNSAT tests, etc. Only a handful of tests in part 1 resulted in solver times that were 4s or longer, and thus for part 2, if I want to properly compare how well the different versions of the solvers perform in regard to each other, I will need to create longer and harder problems.

To create harder problems that will take a good amount of time to solve, I examined the instances in part 1 that took longer to run (for example, intance10, instance13, instance14), and tried to understand why they resulted in the MiniZinc solver taking more time. I came to the conclusion that these instances simply had a larger value for N (i.e. number of pizzas) and the value of M (number of coupons) was about half of N. In addition, when compared to the other instances where I am testing different types of edge cases and so on, these few instances had more realistic numbers and the numbers were all within a closer range to each other.

Thus, after examining a select few of the harder instances from part 1, they seem to appear harder and take more time for the solver to run, because they are larger problems, and they have more realistic numbers for pizza prices, and the number of pizzas needed to get some number of pizzas for free. Based on this, for part 2 I decided to create 15 tests instances that tried to resemble real life scenarios. Meaning, we have a large number of pizzas we want to buy, and we have about n/2 coupons to use. The prices of pizzas will vary, but will usually be in the range of $5 - $35, and the number of pizzas that are needed to be bought before we can get free pizzas will be more realistic and reasonable (e.g., buy 3 pizzas and get 1 for free, or buy 6 and get 2 for free, etc.).

In an effort to create a family of realistic test instances, I ran into the problem of having very large run times that resulted in waiting for over 40 minutes for some instances with large values for N. Thus, to have reasonable run times, I limited my test instances to be of size N = 14, and M = 7. I found having realistic variations of the problems with these sizes provided a good set of problems to test my solvers on, as I got a wide variety of minimal costs, and a wide variety of run times (run times in the range of a couple seconds, to 10 minutes, to 45 minutes for one of the test instances).

EXPERIMENT DETAILS

This experiment entails improving the performance of the MiniZinc solver. Earlier on I purposed two solutions that should help in achieving improved performance.

1. Modifying our specification by adding redundant constraints.
2. Leaving the constraints as they are, and executing the MiniZinc solver with some non-default parameters.

In order to execute MiniZinc with some non-default parameters, I need to run MiniZinc via the Linux command line. To install MiniZinc via the command line, run the following steps (assuming you are working with a Linux system)

1. sudo apt-get update
2. sudo apt-get install
3. sudo apt install minizinc

MiniZinc has a wide variety of backend solvers that it can use, such as Gecode, Chuffer, and OR-Tools to name a few. In this experiment, MiniZinc will be run with the Gecode backend solver (as we did in part 1 of this project).

To execute the MiniZinc file containing our constraints, on some data set, we simply run

- sudo minizinc --solver Gecode pizza.mzn instance1.dzn

The first question I asked, was can we improve the performance of MiniZinc on our problem instances, by adding additional constraints. Since our specification of the *Pizza* problem, which contains 9 constraints as mentioned in part 1, is already enough to correctly specify the problem, I needed to come up with additional redundant constraints that may speed up the process.

I believed that adding redundant constraints should in fact result in improved solver time due to the experiment performed in Assignment 2 which was about Quasigroup Completion. Redundant constraints for those types of problems resulted in improved performance, so it could be the case for the *Pizza* problem as well.

I added 2 redundant constraints to the 9 constraints that were already specified in the problem. The 2 new constraints are defined as follows.

1. If you didn't use a coupon, no pizzas are used to justify using that coupon.
2. If we got a pizza for free, only 1 coupon was used to get that pizza for free (i.e., we do not apply another coupon to an already free pizza)

The first redundant constraint was modelled in MiniZinc as follows.

- constraint forall(coupon in Coupons)(not exists(pizza in Pizzas)(UsedFor[coupon,pizza]) -> not exists(pizza in Pizzas)(Justifies[coupon,pizza]));

Which can be read as "for all coupons, if there is no pizza that is gotten for free by using that coupon, then for all pizzas, no pizza is used to justify using that coupon."

The second redundant constraint was modelled in MiniZinc as follows.

- constraint forall (p in Pizzas)(forall(c1 in Coupons)(UsedFor[c1,p] -> not exists(c2 in Coupons where c1 != c2)(UsedFor[c2,p])));

Which can be read as "If pizza p was gotten for free by apply coupon c1, then for all other coupons c2, coupon c2 is not used to get pizza p for free".

These 2 constraints are redundant, and are not required to specify the problem, as the original 9 constraints are enough, but adding these redundant clauses should speed up the solver time in coming to a conclusion about the minimal cost.

In addition to trying to achieve faster solving time with redundant clauses, the second question I asked was can we improve solver time by tweaking the parameters that the MiniZinc solver uses. There are different parameters that MiniZinc uses that can be tweaked, and below is a list of different options that were considered

- Using a different back-end solver such as Chuffed
- Using different "flattener-input" options that would change different aspects of MiniZinc compilation
- Flattener two-pass options

Out of these 3 options that can be used to improve solver performance, I found the "Flattener two-pass" options to be the most interesting. From MiniZincs online documentation, two-pass compilation is defined as follows.

> *The MiniZinc compiler will first compile the model in order to collect some global information about it, which it can then use in a second pass to improve the resulting FlatZinc. For some combinations of model and target solver, this can lead to substantial improvements in solving time. However, the additional time spent on the first compilation pass does not always pay off.*

Essentially we first compile the MiniZinc code to get some information about the

problem, and then we can use this information and re-compile on a second pass which can better optimize the resulting compiled code (i.e. the FlatZinc).
There are different options of this two-pass method that MiniZinc supports, but the option that I went with for this experiment is called “shave”. What “shave” does is it will probe bounds of all variables at the root node and will perform root-node-propagation with Gecode.

As mentioned in the previous section, this experiment will be performed on 15 test cases that were created to take a good amount of time to solve, as they contain large values for N, a 2:1 ratio between the number of pizzas and the number of coupons, as well as realistic values for pizza prices, the values needed to use a coupon, as well as the number of pizzas obtained for using a coupon. This set of large and realistic problems should prove to take some time for the solver to solve, as there are many different combinations of buying pizzas and using coupons due to the large N value, as well as the fact that by making the problem realistic, we introduce a tighter range of values, and thus we remove any outliers that can result in very easy decisions for the solver (eg. If a pizza cost $100, it will likely not be gotten for free, and if a coupon requires 100 pizzas be bought, that coupon will likely never be used.) This tighter range of values helps to avoid easy decisions.

Solver performance was measured by the time taken to output the minimal cost for the problem instance. Since these instances took more time than the instances from part 1, I performed 2 runs for each problem (rather than 3) and took the average. Each problem was timed using the Linux “time command”. The time command outputs 3 different times, real, user, and system. The “real” time was recorded, which is the wall clock time (i.e., the time from start to finish of the command). Thus, each problem was timed with the following command.

```
time minizinc --solver Gecode pizza.mzn
instance1.dzn
```

RESULTS

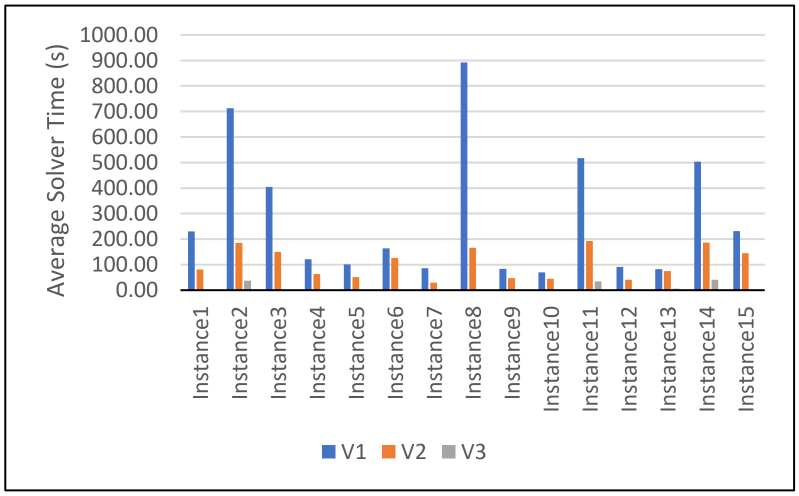
After performing all 15 test cases with the 3 different solving techniques, the results are quite clear. We considered 3 different soling techniques as mentioned previously to improve solver time for MiniZinc.

- Version 1 (**V1**): An unmodified MiniZinc specification, with default settings and the original set of constraints defined in part 1. This version of the solver will serve as the base to which we compare the other versions to.
- Version 2 (**V2**): A modified MiniZinc specification where we added 2 redundant constraints.
- Version 3 (**V3**): A unmodified MiniZinc specification with the original set of constraints defined in part 1, but the solver will be run with the “shave” parameter (a non-default parameter)

On the next page is a table of the collected averages (in seconds) for each of the instances that were executed by the 3 versions of the solver. Each problem instance was run 2 times, and the average value was taken. However, instance 16 was only run once with V1, as it took 45mins to run, and there was not enough time to run it again.

	V1	V2	V3
Instance1	230.55	80.96	1.20
Instance2	712.58	184.60	37.41
Instance3	403.89	150.36	0.80
Instance4	121.35	63.51	0.72
Instance5	101.47	51.50	0.82
Instance6	163.22	125.59	1.15
Instance7	85.57	29.59	0.63
Instance8	891.68	166.08	0.75
Instance9	83.33	46.73	0.81
Instance10	69.41	45.19	0.82
Instance11	516.79	192.47	35.09
Instance12	91.57	41.17	3.51
Instance13	82.05	75.19	6.01
Instance14	503.23	185.94	41.25
Instance15	231.46	145.07	3.66
Instance16	2718.34	577.31	383.38

As we can see from the above data, the unmodified MiniZinc specification performed the slowest out of the three, with the MiniZinc specification with redundant constraints coming in as the second fastest, and the MiniZinc specification with the non-default “shave” parameter being the fastest by a large margin. To make the differences between the different solvers clearer, below is the data table presented in a bar chart.



From the bar chart, we can clearly see how each of the solvers performed in relation to each other. We can see that V1 (unmodified MiniZinc) took much longer to solve problem instances than V2 (MiniZinc with redundant constraints) and V3 (MiniZinc with the “shave” parameter). V2 was able to find the solution in way less time than V1, however, V3 executed all problem instances in less than a minute (except for instance 16), and far outperformed both V1 and V2. For example, looking at instance 8 we can see from the table, that V1 took 891s (~15min) to find the optimal solution, while V2 took 166s (~3min), and V3 took less than a second (0.75s).

From a solver time of 15min, down to 3min, and down to less than a second, it is clear that adding additional constraints can improve solver time, but using the different non-default parameters that MiniZinc provides is a much better solution. In the bar graph, I chose to exclude instance 16, which was an extreme problem instance, and leaving it in would make the graph illegible. In instance 16, V1 took 2718s (~45mins) to find the optimal solution, V2 took 577s (~9.5mins) and V3 took 383s (~6mins). Once again we can see the major improvements that can be made to the MiniZinc solver by adding additional constraints, or by using some of the non-default parameters it has to offer.

DISCUSSION

After performing the experiment, I can conclude that my original two questions have been answered. The two questions I had originally were

1. Can we modify and/or add redundant clauses to our specification to improve solver performance?
2. Can we adjust and or modify the parameters that MiniZinc utilizes to improve solver performance?

Questions 1 was proven to be true, as adding two redundant constraints to the original problem specification improved solver time tremendously, resulting in the solver taking half the time, or often less than half the time, that the original specification with the 9 constraints took.

Question 2 was also proven to be true, as executing MiniZinc with the original specification, but with adding the extra “shave” parameter which performs some pre-processing, resulted in even better performance when compared to the specification with the redundant constraints.

These results helped me understand that while one can correctly model a problem using declarative solving tools such as MiniZinc and come to a solution for their problem, there is a lot of extra things the user can do to improve the performance of their chosen solver. Adding redundant constraints looks to be a good choice in speeding up solver performance, but looking into solver specific parameters seems to be a better options, based on my experiment.

Before performing this experiment, I thought declarative problem solving only involved a user specifying their problem using some tool, and then they can run their tool on the problem to get a solution. However, there a lot of parameters the user can tweak, that I did not know about prior to this experiment. For example, the different pre-processing steps MiniZinc can perform, the different steps it can take during compilation, the different backend solvers that it can use, all greatly affect the solvers performance. Thus, I learned that in declarative problem solving, it is important to be aware of the different parameters the solver allows you to tweak, as it can greatly affect performance. Had I not known about the different parameters that MiniZinc can set, such as the “shave” parameter I used, I would have thought that the IDP system I used in part 1 outperforms MiniZinc every time. However, after tweaking some of the MinZinc parameters, I can see that this is not necessarily the case.

If I wanted to continue this study in the future, I would look more deeply into the parameters that MinZinc provides and how they work, such as the difference between the Gecode and Chuffed backend solvers, the effects that the different compiler and pre-processing options have between each other, and I would also see if other tools such as IDP provide similar parameters and see how they compare.

APPENDIX

Four folders are provided, each containing different files.
Part 1 – Minizinc

- **pizza.mzn**: The specification file for the *Pizza* problem in MiniZinc for part 1 or the report.
- **instance.dzn**: 20 MiniZinc test instances for part 1 of the report. Note, MiniZinc and IDP are using same tests instances, just the syntax of the files for MiniZinc and IDP are different.

Part 1 – IDP

- **pizza.idp**: The specification file for the *Pizza* problem in IDP for part 1 or the report.
- **instance.idp**: 20 IDP test instances for part 1 of the report. Note, MiniZinc and IDP are using same tests instances, just the syntax of the files for MiniZinc and IDP are different.

Part 2 – MiniZinc

- **pizza.mzn**: The specification file for the *Pizza* problem (as an optimization problem) using the original set of constraints, for part 2 of the report.
- **pizza2.mzn**: The specification file for the *Pizza* problem (as an optimization problem) including the redundant constraints, for part 2 of the report.
- **instance.dzn**: 15 MiniZinc test instances for part 2 of the report.

Data

- **Data–Part1**: Collected data for part 1 of the report.
- **Data-Part2**: Collected data for part 2 of the report.