CS4402 - P1

The Operations Room

200007413

October 23, 2023

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Aims and Objectives

The main aim for this practical was to integrate my understanding of constraint modelling with a realistic scenario. Additionally, to evaluate models with with regards to efficiency, correctness, and clarity.

The objectives for this practical were to: develop a constraint model in the Essence Prime Modelling language for use with the SavileRow constraint assistant software; document the process of design; and then evaluate the result.

1.2 Approach

I followed an incremental design approach to this implementation. Starting from the simplest implementation of the constraints, I added each set of constraints gradually, whilst testing using the smaller instances given. Once the fully working model was created, I then considered re-designing sets of constraints to optimize the model for the larger instances.

Finally, I considered the ways to change the SavileRow solver through command line parameters, to further optimize solving-times.

1.3 Report Structure

The first section of this report will discuss the process of designing and implementing the constraint model - this will include a discussion of the translation from problem to variables, domains, and constraints, the constraints implied from the problem, as well as how the output translates to the higher-level solutions to the problem.

Additionally, there will be a review of the optimisation process such as the symmetry-breaking constraints added, and the decision to add a `total_duration` search heuristic which greatly improved search times.

Following this, I will evaluate the final model - giving the search times for each given instance; and considering where the model is well-optimized, as well as redundancies which slow the solver down.

Finally, I will discuss the further optimisations to SavileRow which can significantly improve the search times for the largest models. This includes using a SAT representation for heuristic searching; and a comparison of the different preprocessing options, and backends for SavileRow which are changed through the command line.

Chapter 2

Design and Implementation

2.1 Input Domains and Ranges

The first step in creating a constraint model for this problem was to create a high-level overview of the variables and constraints required. This was particularly helpful, as it enumerated all of the basic elements of the model, which could then be incrementally added and tested.

Before adding in the input variables, I considered their domain. There are some input variables whose domains are just positive integers, or 0, as they setup the instance constraints. These are the number of: surgeons, operations, and specialities. The other input value domains are restricted by those values:

The maximum number of ordering restrictions is technically restricted by the number of operations. As each operation can only have a relation to another operation once (not including itself). Therefore, the maximum domain can be is follows the triangle numbers (where n is the number of operations):

$$\frac{(n-1)^2 + (n-1)}{2}$$

However, I decided not to implement this restriction, as if the size of the ordering list is greater than this value it either means: there are redundant ordering relations, which doesn't affect the working of the solver except potentially slow it down; or it means conflicting ordering relations are present which will make an instance unsatisfiable which may be useful in this scenario as it would represent legal/practical catch-22s which should be identified.

The list of duration's and manning indexes based on the operation, and so it was useful to create a domain Operation, which had the range 1 to the maximum operations.

Then, as the list of ordering relations is limited by the given number, it was useful to create a domain Ordering with range from 1 to the number of ordering relations.

As each entry in the list of Manning's is indexed by the speciality index, it was useful also to create a Speciality domain with range 1 to the maximum speciality.

The range of the values for duration's for each operation was determined to be a positive integer including zero, as an operation cannot take less than no time.

The range of values for available_personnel for each speciality is the positive integers. It doesn't matter if this includes zero or not as it doesn't affect the correctness of solutions, but if any operations require that speciality at all then the instance won't be solvable.

The order list have two values each in the range of the Operations domain.

The specification required that the start times, end times, and main surgeons were required for the scenario, and so this was a good place to start for the decision variables.

Therefore, it was useful to create a new domain for the set of times. This problem technically has an infinite domain for the times. However, it was easy to come up with a worst-case scenario to convert the problem to a finite time domain. As long as no operation requires more speciality staff on it's own than is available, and there is at least one surgeon (inferred constraints inherent to the problem), we can solve the problem in a finite time domain by doing surgeries one-by-one. This has a worst-case time of the sums of each duration - this was used to create the Time domain, with a minimum of 1.

2.2 Decision Variables

The specification gave an example solution with the start times, end times, and main surgeon for each operation, and so this seemed a good place to start for the decision variables.

The start times and end times arrays are 2D sequences with the times for each operation given indexed by the operation number, with the value domains from 1 to the worst-case calculated before.

2.2.1 Optimisation Heuristic

To begin with, the optimisation could be done by adding the search heuristic:

```
minimise max(end_time)
```

This requires searching the list of end_times for the maximum value for each set of end_times searched, which is a lot of operations given larger instances.

To prevent this, another decision variable can be added which will work as the heuristic, this is `total_duration`. This is an integer value representing the total time for a solution, and has the domain of the maximum duration input value, and the worst-case value. This will require later a constraint which can easily restrict the domain of the end times down based on this value.

Using this new decision variable as the minimising factor, we can gradually increase the maximum end_time value as values are invalid, to minimize the total duration in an optimised order. This variable was also set as the branching variable through:

```
branching on [total_duration]
```

This tells the solver to focus on looking for the total duration in ascending order. This is what we want as we are looking for the first value which gives an solution (by definition the smallest total duration).

2.2.2 Implied Constraints

The problem has some constraints that are implied by the specification. These were the first constraints to be implemented as they are mostly very simple.

First, the answer to the problem should be a timetable of the start and end times for each operation in a day. Time only moves forwards, and so the constraint that the start time should be before the end time is implied:

```
$ Symmetry break the start time to prevent valid switching end and
    start orders
forAll i : Operation . i < n_op ->
    start_time[i] <= start_time[i+1],
```

A constraint from the model comes as we have decision variables for the start and end times. The input contains the duration for each operation, and so it is technically not necessary to have an end time decision variable, as it can be calculated by adding the duration to start time. However, having to calculate this end time every time you need it for an operation is costly when solving for each item in the domain, and so adding an implied constraint to the end times is easier:

```
forAll i : Operation .
    $ Apply the end_time constraint
    start_time[i] + duration[i] = end_time[i],
```

Another constraint inherent to the model is given because we have a total duration variable, but the end times have a domain maximum of the worst-case. It is implied that none of the operations end times exceed the total duration.

```
forAll i : Operation .
    end_time[i] <= total_duration,
```

2.3 Constraints

The easiest constraint to add initially is the ordering constraints, as it is just a list of tuples where the first must finish before the second starts.

```
$ Apply the simple ordering relationships
forAll o : Ordering .
    end_time[order[o,1]] <= start_time[order[o,2]],
```

Next, it is relatively easy to check the main surgeon conflicts through the main surgeon list. By checking each pair of operations, we can check if their times overlap, and then if they are make sure their surgeons are different.

```
$ Check main surgeon conflicts
forAll o1,o2 : Operation . ((o1 < o2) /\
    $ Check if they conflict in time
    start_time[o2] < end_time[o1] /\ end_time[o2] >= start_time[o1
       ]) -> (
    main_surgeon[o1] != main_surgeon[o2]
    ),
```

By checking that $o1 < o2$ first, we are making sure that we are only checking pairs once (and not their reverse), reducing redundant checks.

The final constraint required for a correct solution is the specialist personnel availability checks. The first solution was to loop over each unit in the time domain, then sum the manning requirements for each operation running through that time:

```
$ Specialist personnel constraints
forAll t : int(1..total_duration) .
    forAll s : Speciality .
        sum([manning[i,s] * (start_time[i] <= t /\ end_time[i]  >
           t) | i : Operation]) <= available_personnel[s],
```

This works, but becomes very expensive as the instance becomes more complex, and the instance gets closer to the worst case, as more and more time intervals must loop through all the different operations. It is also very redundant, as if there are no changes in operations, then we are checking the same values over the duration of operations.

Later, it was considered that this can be optimised significantly by thinking about the important times to check availability. This is when there are changes to the operations running - specifically only when operations start (as if an operation doesn't exceed availability, it ending will only reduce requirements). Therefore, a more optimised version was added:

```
$ Check specialist personnel availability
$ Requirements can only change at each operation start time so we
   can just check those times
forAll t : Operation .
    forAll s : Speciality .
        sum([manning[i,s] * (start_time[i] <= start_time[t] /\
           end_time[i]  > start_time[t]) | i : Operation]) <=
           available_personnel[s]
    ,
```

2.3.1 Symmetry Breaking Constraints

The first symmetry is inherent to the problem, and is the order of operations can be reversed and still be valid. To break this symmetry, a simple constraint:

```
forAll i : Operation . i < n_op ->
    start_time[i] <= start_time[i+1],
```

This makes sure that the start times for each operation is in increasing order based on their index.

Another inherent symmetry is a value symmetry: Once the surgeon numbers for each surgery have been set, they can be arbitrarily swapped and still be valid. Therefore, it is necessary to implement an ordering constraint to break this. This can be done by making sure that when operations overlap, the main surgeon is not just checked as different, but less than the second (changing from the previous !=).

```
$ Check main surgeon conflicts
forAll o1,o2 : Operation . ((o1 < o2) /\
    $ Check if they conflict in time
    start_time[o2] < end_time[o1] /\ end_time[o2] >= start_time[o1
        ]) -> (
    main_surgeon[o1] < main_surgeon[o2]
    ),
```

2.4 Converting Output to Solutions

The output from this model is very simply converted to a solution to the higher-level problem. If unsatisfiable, then there is no solution. Otherwise, the output will give a start time list and end time list indexed in the same order of operations as given in the duration and manning lists from the input. The times are in the unit given in the input (can be any unit), and 1 represents the start time. There is also a list of main surgeons indexed in the same operations order. The main surgeons list gives the surgeon by index, but which surgeon is which index does not matter as long as they don't change.

Chapter 3

Evaluation

This section contains an evaluation of the final model submitted. There are graphs and tables with runtimes for each instance, as well as tables and charts comparing different command line options. For ease of graphing, the instances have been numbered 1.1-4.10, where small/1.param is 1.1, medium/6.param is 2.6, and very_large/10.param is 4.10.

3.1 Runtimes

Overall the final model was very efficient at solving most of the problems. As table A.1 and fig. 3.1 show, with the default SavileRow configuration, it compiles and solves all of the small and medium sized instances in under a second. It compiles all but the last two very large cases in the 60 second time limit, and solves 70% of the large instances in under 60 seconds.

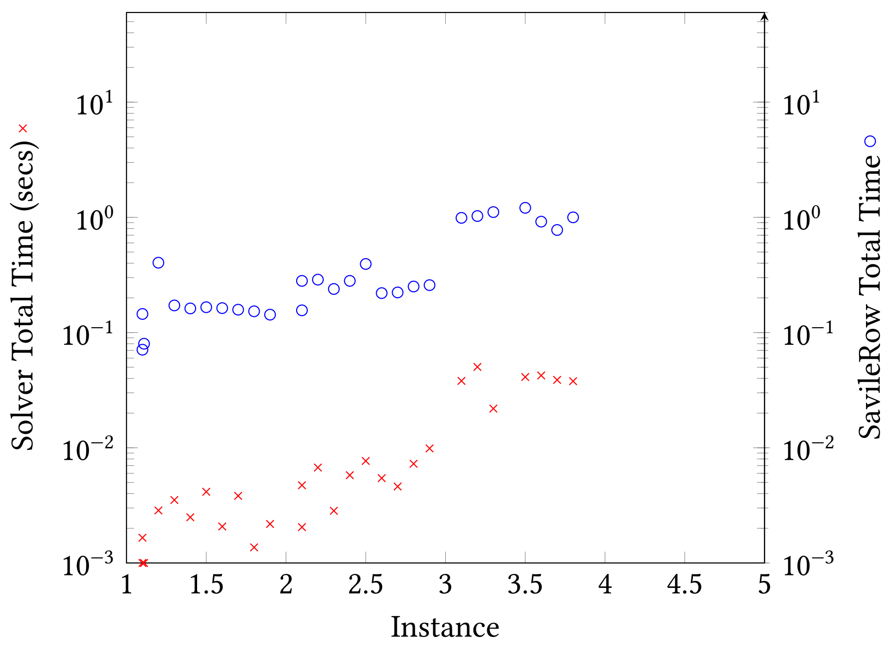

Figure 3.1: Graph to the total time taken for SavileRow preprocessing and solving times (logarithmic, and +0.001 to prevent 0 value error) for each instance (filtering out unsatisfied instances).

3.2 SAT Encoding

One of the biggest improvements to the solver that was found was to introduce SAT encoding for variables, which does not change the model, but the way SavileRow compiles the instance. The greatest change for this was to take advantage of the search heuristic `total_duration`, as it repeatedly checks the midpoint for the upper and lower bound for this value, and then checks the midpoint of which either side results in a smaller value.

When compared to the same model ran without SAT encoding (see. fig. 3.1 and table A.1), the resulting solver times are substantially improved (see. fig. 3.2 and table A.2) - only timing out on the final 2 largest instances. The SavileRow times are not significantly affected by this encoding, but allows some of the smaller instances to be solved quicker than the accuracy of the SavileRow logging system (aka. 0 time).

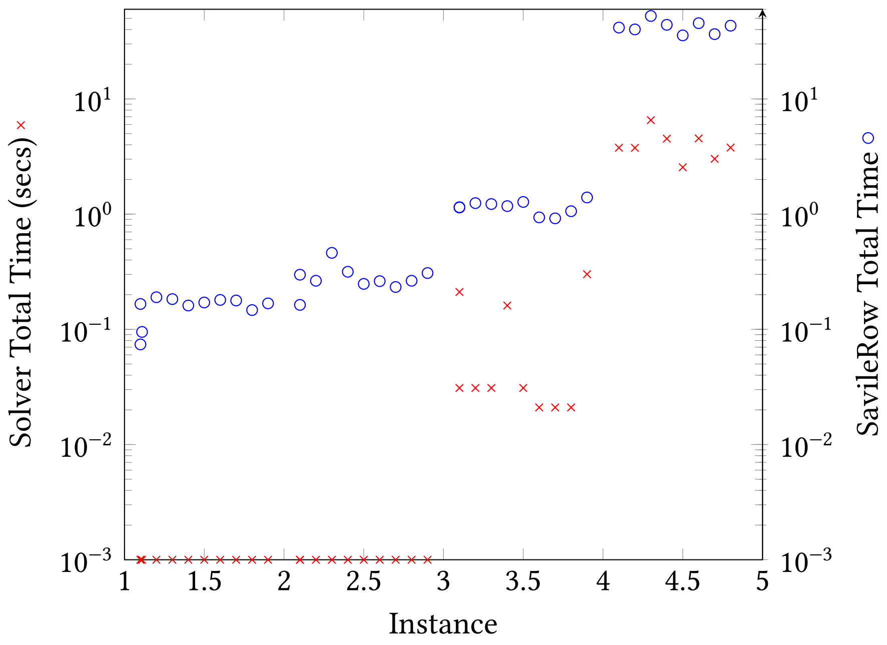

Figure 3.2: Graph to show the preprocessing and solve times (logarithmic, and + 0.001 to prevent 0 value error) for each instance for the final model with SAT encoding (filtering out unsatisfied instances).

However, the outliers with higher times than similar solve or compile times than similar sized instances are mostly the same, (aka. 2.4) which indicates that there is still room for improvement.

3.3 Factors Affecting Runtime

It seemed clear that one of the main factors in the time taken to solve the problems was the number of solver nodes that were generated by SavileRow, which are the number of tree nodes to search through after compilation. fig. 3.3 shows that whilst there is a strong positive correlation between the number of solver nodes and the compile and solver times, some outliers are still present with low solver nodes, but high solver times.

This may suggest that there may be some additional implied or symmetry constraints that haven't been implemented which would make searching easier. However, there will always be worst-case scenarios, and they only take around a second longer to solve than similar sized instances.

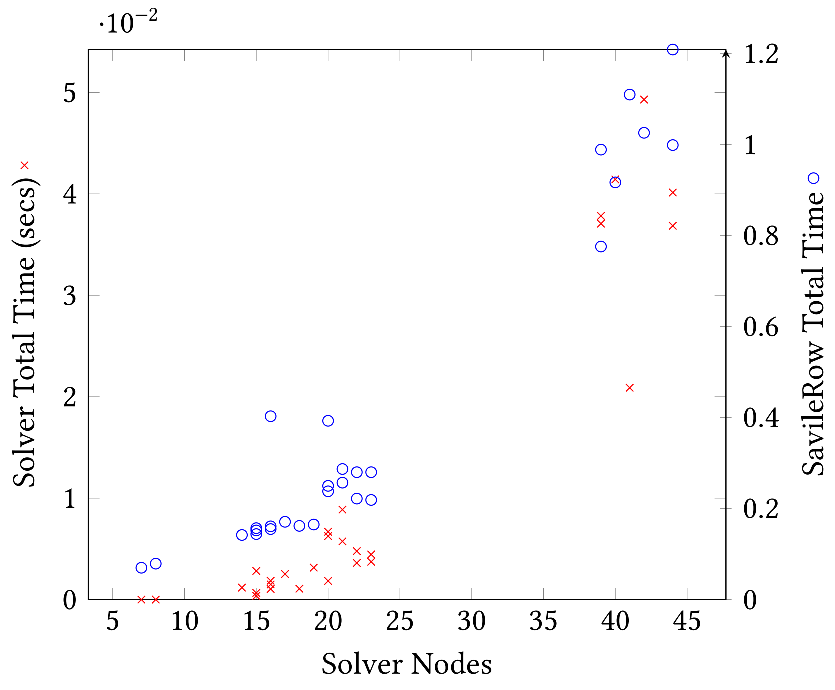

Figure 3.3: Graph to show the relationship between the number of solver nodes on both the total time the solver took to solve each instance, and SavileRow preprocessing time (filtering out unsatisfied instances).

One interesting piece of information is shown in fig. 3.4, which shows the expected trend of higher savile row compilation times correlating with the solver times, but the colouring shows that the solver nodes seems to have a greater impact on the solver time than the compilation time, as some smaller instances take as long as larger instances to compile, but a clear increase in size to solver time is present.

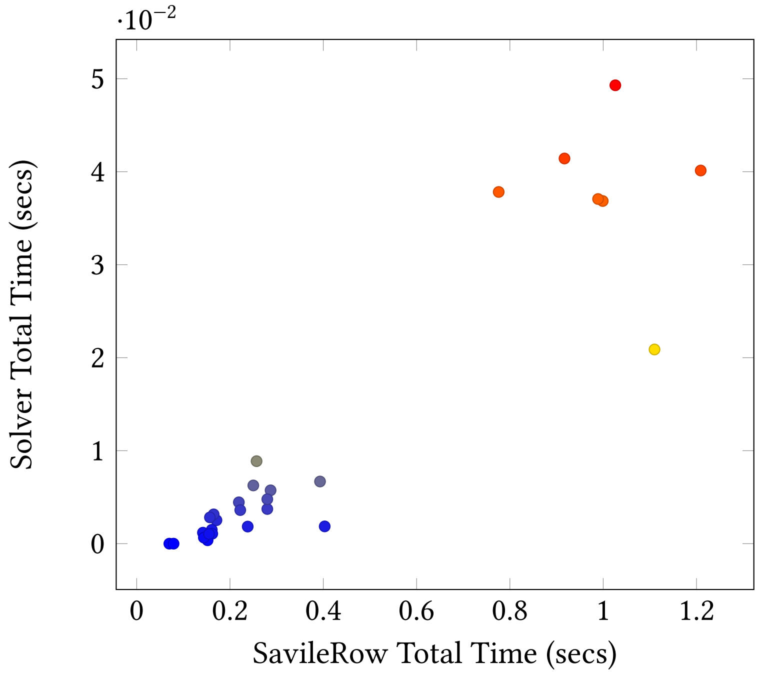

Figure 3.4: Graph to show the relationship between the SavileRow preprocessing time and the total solving time (filtering unsatified instances). Colouring indicating the number of solver nodes.

When using SAT encoding, SavileRow converts some of the solver nodes to SAT variables, when looking at the relationship between solve times and number of SAT vars, we see a cleaner relationship between the size and the compilation time (see. fig. 3.5), especially when looking at times greater than 1 second (see fig. 3.6), as there is a proportionally smaller impact from extraneous computation speed factors.

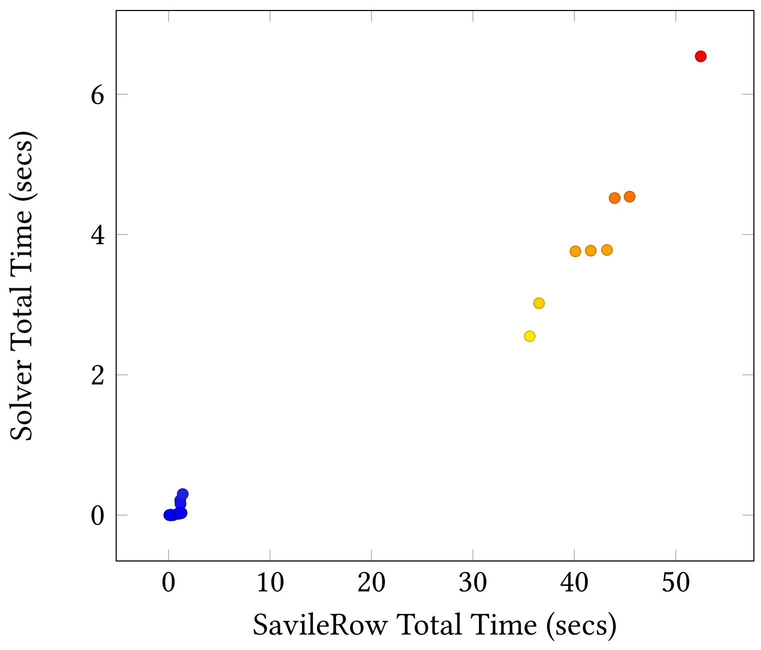

Figure 3.5: Graph to show the relationship between the SavileRow preprocessing time and the total solving time (filtering unsatisfied instances). Colouring indicating the number of SAT vars.

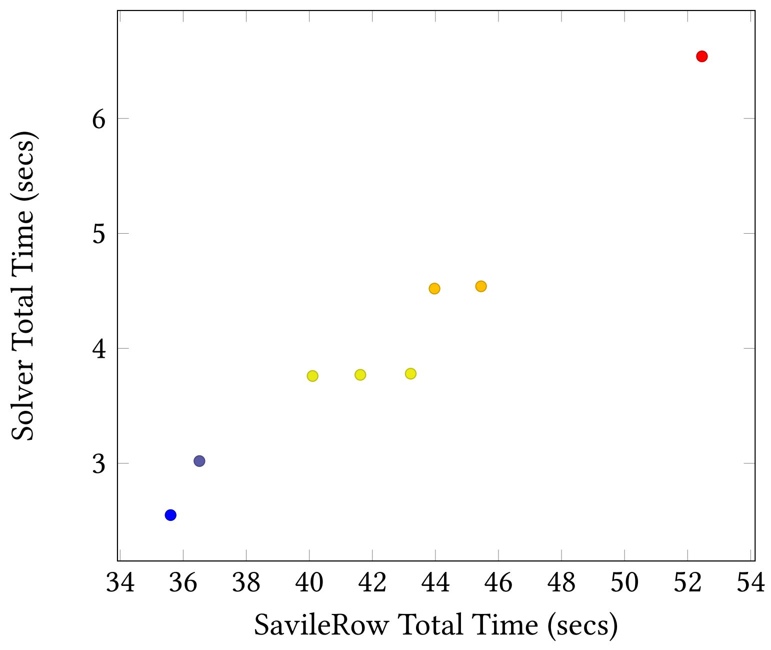

Figure 3.6: Zoom in on SAT encoding relationship from fig. 3.5 to SavileRow Total Time region 35s to 60s, showing a linear relationship between the times taken to process, and to solve.

Chapter 4

Conclusion

Overall, one of the largest lessons learned from this experience has been the importance of search ordering for optimisation problems, and making sure that you have picked a good search heuristic. The SAT searching method sped up the search dramatically.

Additionally, the importance of considering when to do calculation, such as the end_time calculation being a decision variable instead of calculated on the fly to prevent repeated calculation was really important for search optimisation.

I feel that the aims of this practical were met well with this submission through meeting all of the objectives: the submission contains a correct and effective constraint model, along with a detailed and extended evaluation of the design choices, as well as further research into the topic for an even more effective solution.

Appendix A

Model Runtime Data

Table A.1: Table showing SavileRow runtimes for each instance for the final model using the default SavileRow setup. Ran on lab machine pc1-0350-l on October 23, 2023. A 60 second time limit was placed on the solver, and 120 second limit on savile row (which includes solver and preprocessing) for each instance.

Size	Instance	Solved	Nodes	SR Time (secs)	Solver Time (secs)
small	1.1	1	7	$7 \cdot 10^{-2}$	0
small	1.2	1	16	0.4	$1.86 \cdot 10^{-3}$
small	1.3	1	17	0.17	$2.52 \cdot 10^{-3}$
small	1.4	1	16	0.16	$1.49 \cdot 10^{-3}$
small	1.5	1	19	0.17	$3.15 \cdot 10^{-3}$
small	1.6	1	18	0.16	$1.07 \cdot 10^{-3}$
small	1.7	1	15	0.16	$2.83 \cdot 10^{-3}$
small	1.8	1	15	0.15	$3.66 \cdot 10^{-4}$
small	1.9	1	14	0.14	$1.18 \cdot 10^{-3}$
small	1.1	1	15	0.14	$6.56 \cdot 10^{-4}$
small	1.11	1	8	$7.9 \cdot 10^{-2}$	0
medium	2.1	1	23	0.28	$3.73 \cdot 10^{-3}$
medium	2.2	1	21	0.29	$5.74 \cdot 10^{-3}$
medium	2.3	1	20	0.24	$1.84 \cdot 10^{-3}$
medium	2.4	1	22	0.28	$4.78 \cdot 10^{-3}$
medium	2.5	1	20	0.39	$6.69 \cdot 10^{-3}$
medium	2.6	1	23	0.22	$4.45 \cdot 10^{-3}$
medium	2.7	1	22	0.22	$3.61 \cdot 10^{-3}$
medium	2.8	1	20	0.25	$6.26 \cdot 10^{-3}$
medium	2.9	1	21	0.26	$8.87 \cdot 10^{-3}$
medium	2.1	1	16	0.16	$1.05 \cdot 10^{-3}$
large	3.1	0	$7.13 \cdot 10^{7}$	1.01	60.01
large	3.2	1	42	1.03	$4.93 \cdot 10^{-2}$
large	3.3	1	41	1.11	$2.09 \cdot 10^{-2}$
large	3.4	0	$9.15 \cdot 10^{7}$	1.16	60.01
large	3.5	1	44	1.21	$4.01 \cdot 10^{-2}$
large	3.6	1	40	0.92	$4.14 \cdot 10^{-2}$
large	3.7	1	39	0.78	$3.78 \cdot 10^{-2}$
large	3.8	1	44	1	$3.69 \cdot 10^{-2}$
large	3.9	0	$1.44 \cdot 10^{8}$	1.06	59.96
large	3.1	1	39	0.99	$3.71 \cdot 10^{-2}$
very_large	4.1	0	$4.23 \cdot 10^{7}$	41.09	60.01
very_large	4.2	0	$4.53 \cdot 10^{7}$	38.4	60.01
very_large	4.3	0	$4.41 \cdot 10^{7}$	50.07	60.01
very_large	4.4	0	$6.34 \cdot 10^{7}$	43.42	60.01
very_large	4.5	0	$5.76 \cdot 10^{7}$	34.94	60.09
very_large	4.6	0	$7.76 \cdot 10^{7}$	43.59	60.01
very_large	4.7	0	$5.61 \cdot 10^{7}$	36.15	60.01
very_large	4.8	0	$8.45 \cdot 10^{7}$	43.59	60.01
very_large	4.9	0	0	0	120.06
very_large	4.1	0	0	0	120

Table A.2: Table showing SavileRow runtimes for each instance for the final model using SAT encoding. Ran on lab machine pc1-0350-l on October 23, 2023. A 60 second time limit was placed on the solver, and 120 second limit on savile row (which includes solver and preprocessing) for each instance.

Size	Instance	Solved	Nodes	SAT Vars	SR Time (secs)	Solver Time (secs)
small	1.1	1	0	47	$7.3 \cdot 10^{-2}$	0
small	1.2	1	0	475	0.19	0
small	1.3	1	0	592	0.18	0
small	1.4	1	0	357	0.16	0
small	1.5	1	0	433	0.17	0
small	1.6	1	0	865	0.18	0
small	1.7	1	0	423	0.18	0
small	1.8	1	0	134	0.15	0
small	1.9	1	0	115	0.17	0
small	1.1	1	0	244	0.17	0
small	1.11	1	0	288	$9.4 \cdot 10^{-2}$	0
medium	2.1	1	0	827	0.3	0
medium	2.2	1	0	1,240	0.26	0
medium	2.3	1	0	378	0.46	0
medium	2.4	1	0	943	0.32	0
medium	2.5	1	0	740	0.25	0
medium	2.6	1	0	814	0.26	0
medium	2.7	1	0	615	0.23	0
medium	2.8	1	0	992	0.26	0
medium	2.9	1	0	1,354	0.31	0
medium	2.1	1	0	377	0.16	0
large	3.1	1	2,697	11,448	1.15	0.21
large	3.2	1	0	10,290	1.25	$3 \cdot 10^{-2}$
large	3.3	1	0	12,313	1.22	$3 \cdot 10^{-2}$
large	3.4	1	1,999	10,448	1.17	0.16
large	3.5	1	0	11,302	1.28	$3 \cdot 10^{-2}$
large	3.6	1	0	6,858	0.94	$2 \cdot 10^{-2}$
large	3.7	1	0	6,779	0.92	$2 \cdot 10^{-2}$
large	3.8	1	0	9,538	1.06	$2 \cdot 10^{-2}$
large	3.9	1	4,015	11,551	1.39	0.3
large	3.1	1	0	10,870	1.14	$3 \cdot 10^{-2}$
very_large	4.1	1	34,547	56,563	41.62	3.77
very_large	4.2	1	47,059	60,810	40.1	3.76
very_large	4.3	1	$1.15 \cdot 10^{5}$	73,929	52.46	6.54
very_large	4.4	1	48,913	65,449	43.97	4.52
very_large	4.5	1	20,502	54,130	35.6	2.55
very_large	4.6	1	50,641	72,207	45.45	4.54
very_large	4.7	1	20,179	56,307	36.51	3.02
very_large	4.8	1	22,348	61,324	43.22	3.78
very_large	4.9	0	0	0	0	120.1
very_large	4.1	0	0	0	0	120.1