Juuso Haavisto
CS4402 Constraint Programming Practical 1: Human Resource Machine
Mon Oct 26, 2020

CS4402 Constraint Programming Practical 1: Human Resource Machine

Introduction

The practical for CS4402 consisted of creating a solver mechanism for Human Resource Machine, a game available, e.g., from Steam. Firstly, I want to thank you for the setup for the practical. It was nice to play a game to understand the problem on-hand precisely and use it to enforce the basic understanding of the problem. I wish more practicals would be structured like this.

For the assignment, I initially approached the problem in two similar ways. First, I tried to find as general rules as possible, which might help infer steps. During the first iteration, most of the time was to understand the Savile Row and the Essence Prime language. A relevant problem I ran into was enforcing how I can choose corresponding values from the input and output parameters. I eventually solved this in a somewhat unintuitive way, which is by using the sum operation. For example, to choose the first value in the input parameter, we can first define a constraint that the first step should be an INBOX step. Next, we can iterate over the steps and use the following selector to choose the correct value from the INBOX:

```
forAll s : steps_iter .
    (steps[s] = INBOX ->
        carrying[s] = input[(sum p : int(1..s) . steps[p] =
INBOX)])
    /\
    ...
```

This approach worked fine, which I then extended into a full solution. The model is found from the project root folder in a file called `hrm3.eprime`.

Variables and Domains

I modeled the `find` variables and domains as follows:

```
find steps: matrix indexed by [int(1..MAX_OPS)] of int(
    INBOX, OUTBOX, COPYFROM, COPYTO, ADD, SUB, BUMPP, BUMPM
)
find carrying: matrix indexed by [int(1..MAX_OPS)] of int(
    -1, min(min_o, min_i)..max(max_o, max_i)
)
find registers: matrix indexed by [int(1..num_registers),
int(1..MAX_OPS)] of int(
    -1, min(min_o, min_i)..max(max_o, max_i)
)
```

Informally, `steps` are the instructions which are chosen at any given time. `carrying` is the matrix holding the carried value at each point in time. To elaborate further, in my model when INBOX happens, then from same index at `carrying` you will find n'th input value the value. `registers` is the matrix holding the register values at any given time. Similar logic to assigning registers work, when a COPYTO happens at index `i`, then *some* register value at `i` gets assigned the value of `carried[i-1]`.

I bounded the domains initially from `-1..100`. Still, I later found out that the problems are satisfiable even if they are bounded by the input and output params' max and minimum values. `-1`, on the other hand, is mostly for debugging purposes to and to define that a register value is undefined (i.e., unused) and to ensure that the solver does not accidentally take the default zero value when it is convenient for it.

The reason the variables are matrices is that a variable reassignment makes the case unsat. For this reason, we must carry on the state at each step.

Constraints

Regarding non-obvious constraints, I was able to find some, such as:

a) absurd to have more than num_register of COPYTO or COPYFROM values in a sequence

```
((steps[s] = COPYTO \/ steps[s] = COPYFROM) /\ s > num_registers
->
    (sum p : int(s-num_registers..s) . steps[p] = s) <
num_registers+1)
```

b) OUTBOX is a hard constraint. We must always have as many OUTBOX steps, and there exists at least one INBOX but at most input length INBOXs.

```
gcc(steps, [OUTBOX], [output_length]),
atmost(steps, [input_length], [INBOX]),
atleast(steps, [1], [INBOX])
```

c) Especially the last two lines here, say that whatever is in the non-empty register slot has been seen before in the carrying matrix (I think). Another version I used was using indexes and `exists` (`exists s : steps_iter . steps[s] = registers[r, s] /\ s <= r`), but I wanted to use a single lex command which otherwise stayed an enigma to me. Switching between the two had seemingly no effect on the result, so I guess they are the same operation.

```
steps[1] = INBOX
/\ steps[MAX_OPS] = OUTBOX
/\ forAll r : reg_iter .
    registers[r,1] = -1
    /\ max(carrying) >= max(registers[r,..])
    /\ (registers[1, r] = -1
        \/ registers[1,..] <=lex carrying)
,
```

d) Here, we guarantee that if there are fewer COPYTOs than num_registers, then some register lane must be nil.

```
(sum p : int(1..MAX_OPS) . steps[p] = COPYTO) < num_registers ->
    exists r : reg_iter .
    forAll s : steps_iter . registers[r,s] = -1
,
```

The main decision procedure is as follows:

```
$ if there is no COPYTO yet, and the value is indifferent,
$ then the next step is either inbox or copyto
(((carrying[s] < output[1+(sum p : int(1..s) . steps[p] =
OUTBOX)]
\/ carrying[s] > output[1+(sum p : int(1..s) . steps[p] =
OUTBOX)]))
/\ ((sum p : int(1..s) . steps[p] = COPYTO) = 0) ->
    (steps[s+1] = INBOX
    \/ (steps[s+1] = COPYTO <-> num_registers > 0)))
/\
$ if the carried value is less and there exists a copy to before
$ then the next value can be one of following
((carrying[s] < output[1+(sum p : int(1..s) . steps[p] =
OUTBOX)])
/\ (sum p : int(1..s) . steps[p] = COPYTO) > 0 ->
    (steps[s+1] = ADD
    \/ steps[s+1] = BUMPP
    \/ steps[s+1] = INBOX
    \/ steps[s+1] = COPYTO
    \/ steps[s+1] = COPYFROM))
/\
$ if the carried value is more and there exists a copy to before
$ then the next value can be one of following
((carrying[s] > output[1+(sum p : int(1..s) . steps[p] =
OUTBOX)])
/\ (sum p : int(1..s) . steps[p] = COPYTO) > 0 ->
    (steps[s+1] = SUB
    \/ steps[s+1] = BUMPM
```

```
    \/ steps[s+1] = INBOX
    \/ steps[s+1] = COPYTO
    \/ steps[s+1] = COPYFROM))
/\
$ if the carried value is an output, then the next value should
be outbox
(steps[s+1] = OUTBOX <-> (carrying[s] = output[1+(sum p :
int(1..s) . steps[p] = OUTBOX)])))
```

It is not pretty, but it gets the job done. Comments above define the rules a bit better. I tried to refactor this multiple times, but performance always deteriorated. Anyway, the gist is that we guide the solver only to allow individual decisions to be available when the next output value compares in a certain way to the current carried value. Further, the OUTBOX statement at the bottom could have OR with a COPYTO, but as an optimization it seemed to do just fine even without such possibility.

Example Solutions

By the instructions, the solver found the following solutions:

3.param

```
language ESSENCE' 1.0
$ Minion SolverNodes: 64
$ Minion SolverTotalTime: 0.016888
$ Minion SolverTimeOut: 0
$ Savile Row TotalTime: 0.765
letting carrying be [3, 3, 2, 5, 5, 9, 9, 12, 21, 21]
letting registers be [[-1, 3, 3, 3, 3, 3, 9, 9, 9, 9]]
letting steps be [1, 4, 1, 5, 2, 1, 4, 1, 5, 2]
```

5.param

```
language ESSENCE' 1.0
$ Minion SolverNodes: 2383
```

```
$ Minion SolverTotalTime: 0.106103
$ Minion SolverTimeOut: 0
$ Savile Row TotalTime: 1.04
letting carrying be [4, 4, 8, 12, 12, 12, 24, 24, 5, 5, 10, 15,
15]
letting registers be [[-1, 4, 4, 4, 4, 12, 12, 12, 12, 5, 5, 5,
5]]
letting steps be [1, 4, 1, 5, 2, 4, 5, 2, 1, 4, 5, 5, 2]
```

Empirical Evaluation

For the table, please see the Appendix of this PDF.

Looking at the chart, we can see that all cases except for 12 are solved in under 10 seconds. 12 takes around 100, which makes it a huge outlier. In fact, there is a long sequence of COPYFROMs. These commands are "no-effect" instructions in my model, and a sequence of them means the actual solution is less steps than what is needed. To elaborate, there is a sequence of 8 COPYFROMs, thus we can deduce that the solution can be solved in at least `MAX_OPS-(8-1)` steps. From the chart, we can also see that the hardest parameter files were 10..12. Somewhat interesting is also the unsat problems 18 and 20 which require relatively big node count whereas the other two unsat instances have a node count of 0.

My initial hypothesis was that the difficulty increases the moment there comes the first COPYTO instruction. This is clearly seen in the sudden increase of solver nodes in cases 5 and 6 which have a COPYTO early on: even though the input length is relatively small, the node count and solver time increase on the logarithmic scale very fast.

To test this, I checked the correlation with amount of steps against solver total time. These are also in the Appendix. I also tested the position of the first COPYTO instruction and how it relates to execution time. In the charts, param 12 has been removed as it was an outlier.

While our sample size is quite low, we can see that the amount of steps seems

to be correlated by the amount of time it takes to solve a case. The location of COPYTO seems rather irrelevant.

When removing BUMP+ and BUMP- from the model, the solver time decreases for `5.param` in both node count and execution time. A diff shows:

```
 language ESSENCE' 1.0
-$ Minion SolverNodes: 2383
-$ Minion SolverTotalTime: 0.089161
+$ Minion SolverNodes: 758
+$ Minion SolverTotalTime: 0.032535
 $ Minion SolverTimeOut: 0
-$ Savile Row TotalTime: 0.857
+$ Savile Row TotalTime: 0.611
 letting carrying be [4, 4, 8, 12, 12, 12, 24, 24, 5, 5, 10, 15,
15]
 letting registers be [[-1, 4, 4, 4, 4, 12, 12, 12, 12, 5, 5, 5,
5]]
 letting steps be [1, 4, 1, 5, 2, 4, 5, 2, 1, 4, 5, 5, 2]
```

It is worth noting that the result does not change with my model at this time. I assume this is because the ADD and SUB commands are declared first in my model, so the search defaults initially. As for the reason why the performance increases, I hypothesize that the search space decreases. I believe this view is supported by the info file generated; BUMP+ and BUMP- removed:

```
# CSETopLevel_number = 20
# CSETopLevel_eliminated_expressions = 40
# CSETopLevel_total_size = 180
# CSE_active_number = 154
# CSE_active_eliminated_expressions = 775
# CSE_active_total_size = 1822
```

BUMP+ and BUMP- active:

```
# CSETopLevel_number = 34
```

```
# CSETopLevel_eliminated_expressions = 68
# CSETopLevel_total_size = 306
# CSE_active_number = 171
# CSE_active_eliminated_expressions = 779
# CSE_active_total_size = 1524
```

We can see that when the operations are active, the CSE total size is almost twice as much. This, in turn, during execution leads to slowdowns as there are more paths to travel.

For my self-generated sets (see folder `my-instances`), some of the mishaps that I faced were that my model does not actively try to minimize the number of steps it produces. This could be implemented by checking whether there are as many OUTBOXs in the steps matrix as there are in the output, and if so, then declare everything rest as `-1`. Yet, this can also be noticed when the last commands then to be just COPYTOs or COPYFROMs. I gave a try, and I timed on most self-generated tests. This is the most likely side-effect of my optimizations.

Yet, even from this perspective, I can envision that the problem becomes harder the more outputs there are than inputs: more inputs gives more availability to choose a fine-fitting argument, whereas if the input is constrained, it might be harder to find the sequence by which to produce each fitting parameter. Another way to put is that input can always be tossed out, but an output argument is a hard constraint.

In conclusion, my model is correct by the following logic: all unsat instances are unsat, and all sat instances are sat. All instances also complete under the 600 second time constraint.

I also found something which I do not fully understand. In a previous model, I used sum operation for register indexing. For some reason, I got fewer solver nodes than with a constant variable. But, the sum operation worked under the assumption that the register length is 1, and there exists only 1 COPYTO command. This seems quite counterintuitive to how using a variable is faster than a constant. E.g., in param 8, the speedup was over 7x vs using a constant.

I have attached a copy of this model in the report, named `report-arficats/hrm_sum.eprime.` This made me wonder if my solver approach is simply something the compiler cannot fully optimize.

Further, my model is quite intuitive to me, but maybe unidiomatic to a domain expert. Yet, I found it a positive learning experience for my learning purposes to try to solve by my way and see it working. I also attest that much time went to understanding Essence Prime correctly, which resulted in spending more time to get a working solution than thinking all the possible underlying optimizing constraints that were available. Some of the things I tried but dropped was "absurd" cases like an ADD being followed by SUB and BUMP+ being followed by BUMP-. I also dropped some fancier rules like prioritizing ADD and SUB over the BUMP commands when the absolute integer distance to the wanted was less with those commands than the others.

Overall I found Savile Row and Essence Prime an excellent solution to approach high level SAT problems (i.e., compared to Z3, which I have tried before) and will most likely use it for my projects in the future.

Extensions

I also did some extension work describe below.

Solver variety

I tried running the Savile Row with `-O3` command and noticed that this decreased my node count and solver time by a significant margin. E.g., for param10:

`timeout 600 ./savilerow-1.8.0-mac/savilerow hrm3.eprime -O3 instances/8.param -run-solver` yields:

```
  language ESSENCE' 1.0
-$ Minion SolverNodes: 175020
-$ Minion SolverTotalTime: 7.68422
```

```
+$ Minion SolverNodes: 4237
+$ Minion SolverTotalTime: 0.93328
 $ Minion SolverTimeOut: 0
-$ Savile Row TotalTime: 1.344
+$ Savile Row TotalTime: 4.81
 letting carrying be [4, 8, 5, 10, 10, 11, 12, 12, 24, 24, 12,
13, 14, 15, 15, 30, 30, 30, 60, 60]
 letting registers be [[-1, -1, -1, -1, 10, 11, 12, 12, 12, 12,
12, 13, 14, 15, 15, 15, 15, 30, 30, 30]]
 letting steps be [1, 1, 1, 1, 4, 7, 7, 2, 5, 2, 3, 7, 7, 7, 2,
5, 2, 4, 5, 2]
```

In summary, even though the problem's overall time takes slightly longer, it is evident that the node count decreases by a wide margin. Supposing a production environment for some practical use-case, it would thus be relevant to try to figure out the "tipping point" at which more time should be spent on rewriting instead of solver execution. Similarly, in a case like mine, we can assume that if the compiler can tremendously speed up the execution with compiler instructions, we may deduce that the actual model could use some rewriting.

Regarding different backends, being on a Mac computer, the only different backend functioning correctly on my system was the SAT one. However, the speedups provided by this engine were tremendous: each case gets now solved in around 1 second (excluding translation). For example, case 12, which took 97 seconds with `minion` took with SAT engine as follows:

```
SolverTotalTime:1.42816
SolverMemOut:0
SATClauses:152511
SavileRowClauseOut:0
SavileRowTotalTime:6.552
SolverSatisfiable:1
SavileRowTimeOut:0
SolverTimeOut:0
SATVars:20976
```

That's an over 10x speedup!

Further, we can now be guaranteed our model's completeness, as it successfully finds every satisfiable case satisfiable and every unsatisfiable case unsatisfiable. This also supports my previous view about the implicit constraints we have not modeled: there may exist some better way to write our model, as the SAT engine is so much faster. Bringing the operation time down for each case would mean restructuring our models to provide better the correspondence that the SAT engine finds but which is being missed by `minion.`

Looking at the `.minion` files produced by the SAT engine for some previous iterations gives us a hint which might be the problem: the SAT engine can find much more implicit equivalences in our model, more specifically showed by the amount of `CSE_active_eliminated_expressions` as given in the file header. Here, for one example, `minion` finds `379` of such cases, whereas the SAT engine finds `1206`. If we `diff` the files, we can see that the SAT solver can create more refined interval sets for our program, such as `DISCRETE carrying_00002 #{-1..90}` of `minion` versus `DISCRETE carrying_00002 #{3..4}` of the SAT engine. Thus, we can assume that whatever it is in our model that tries to hint about these constraints is not as precisely inferred in `minion` as it is with the SAT engine. I hypothesize that with further Savile Row experience, I would learn about what causes these implied rules to be missed, thus closing the performance gap between the two engines.

Optimisation

As noted before, I was able to find some surprising constraints, as noted in the **Constraints chapter**. For example, it might not be apparent from the get-go that the cases are solvable after limiting the decision variable domain to match the input and output max values. To demonstrate the effect, let us consider the second parameter:

For case 12 again, without bounding (i.e., from values being accepted from -1..100), we get the `diff`:

```
 language ESSENCE' 1.0
-$ Minion SolverNodes: 175020
-$ Minion SolverTotalTime: 7.68422
+$ Minion SolverNodes: 188568
+$ Minion SolverTotalTime: 15.8615
 $ Minion SolverTimeOut: 0
-$ Savile Row TotalTime: 1.344
+$ Savile Row TotalTime: 1.394
 letting carrying be [4, 8, 5, 10, 10, 11, 12, 12, 24, 24, 12,
13, 14, 15, 15, 30, 30, 30, 60, 60]
 letting registers be [[-1, -1, -1, -1, 10, 11, 12, 12, 12, 12,
12, 13, 14, 15, 15, 15, 15, 30, 30, 30]]
 letting steps be [1, 1, 1, 1, 4, 7, 7, 2, 5, 2, 3, 7, 7, 7, 2,
5, 2, 4, 5, 2]
```

A similar case is the rule which I refactored later to a `<=lex` rule:

```
num_registers = 1 ->
    forAll r : steps_iter .
    exists s : steps_iter .
        registers[1,r] != -1 <-> (
            (registers[1,r] = carrying[s] /\ s < r)
            \/
            (sum p : int(1..s) . steps[p] = BUMPP) > 0 /\ s = r)
,
```

This means that each register value that is non-empty `-1` much exists in the carried value and before the index of such value, but by empirical testing, I found this only to be applicable when `BUMPP` does not exist. Otherwise, the value might be at the same index because the OUTBOX value might be bumped, which in my model then means that the value is at the position in carried and register value. For cases like 5, I observed this optimization to decrease node overhead by 33%.

Even more astonishing is the prementioned sum case. If we can assume that the number of registers and COPYTO values is 1, the node count is further

decreased. The fascinating thing is that modifying the sum to be a constant (i.e., hard-coding the register to be 1 eradicates the optimization. Not sure what is happening here. A similar case was that `forAll` loops with ANDs were faster than with ORs.

Getting Creative!

As for this extension, I propose a multiplication command called MULT. It works the same way as SUB and ADD, but it multiplies the number. This would be nice in cases like `6.param` as shown from the diff:

```
 language ESSENCE' 1.0
-$ Minion SolverNodes: 14728
-$ Minion SolverTotalTime: 0.561308
+$ Minion SolverNodes: 2006
+$ Minion SolverTotalTime: 0.127573
 $ Minion SolverTimeOut: 0
-$ Savile Row TotalTime: 0.813
-letting carrying be [4, 4, 8, 8, 16, 16, 32, 32, 32, 64, 64, 8,
40, 40]
-letting registers be [[-1, 4, 4, 8, 8, 16, 16, 16, 32, 32, 32,
32, 32, 32]]
-letting steps be [1, 4, 5, 4, 5, 4, 5, 2, 4, 5, 2, 1, 5, 2]
+$ Savile Row TotalTime: 0.926
+letting carrying be [4, 8, 8, 8, 16, 24, 32, 32, 8, 64, 64, 5,
40, 40]
+letting registers be [[-1, -1, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8,
8]]
+letting steps be [1, 1, 4, 3, 5, 5, 5, 2, 3, 9, 2, 1, 9, 2]
```

Here, rule 9 is the multiplication. It could thus reduce the time taken to integer operations in some cases.

While the data vouches for new semantics, I am a rather stubborn person. This is the reason I vote against it. One of main takeaways for me thus far has been that constraints facilitate creative problem solving by trying to understand the problem better.

Conclusion

SAT solvers like Savile Row, introduce a new kind of programming paradigm that differs significantly from imperative programming. Instead of programmer defining state transitions, the constraints under which these state mutations may happen are instead declared. To a layperson, this might make SAT solvers sound like an AI method since the program figures out the answer "by its own."

Yet, the main challenge, at least to me, was the ability to rethink my programming model away from imperative programming towards constraint programming. This was the central part that I consumed time with. Further, as with DSLs in general, I ran into cases in which my solution's performance varied greatly depending on what, to me, seemed like only a slight adjustment. It's worth noting that this is not a problem in the SAT approach in itself, but merely a general compiler problem in which the underlying and more theoretical aspects are attempted to be abstracted away.

Nonetheless, I created a working solution for each case, proving empirically that my model is indeed correct. There were no cases in which the program timeouts without optimizations. With optimizations and especially the SAT solver engine, my model performs almost imminently to all given problems.

Per request, the table data in the added Appendix is also found from the project root folder in a CSV file called `solvertimes.csv.`

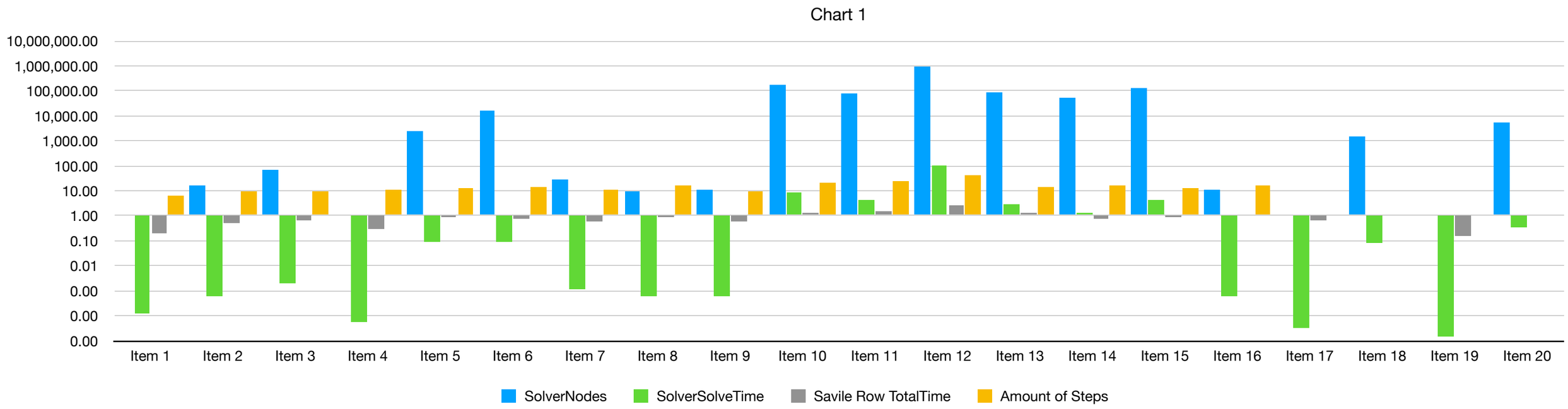
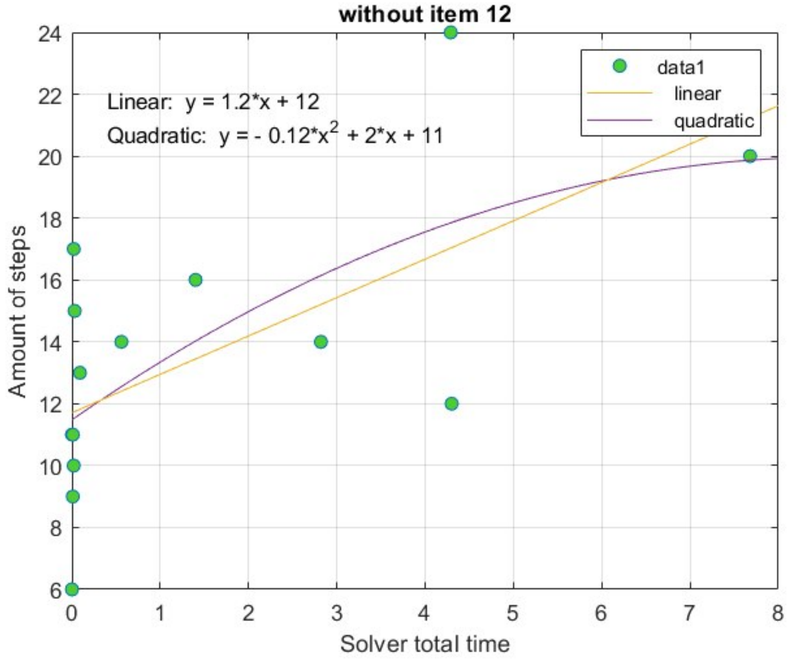
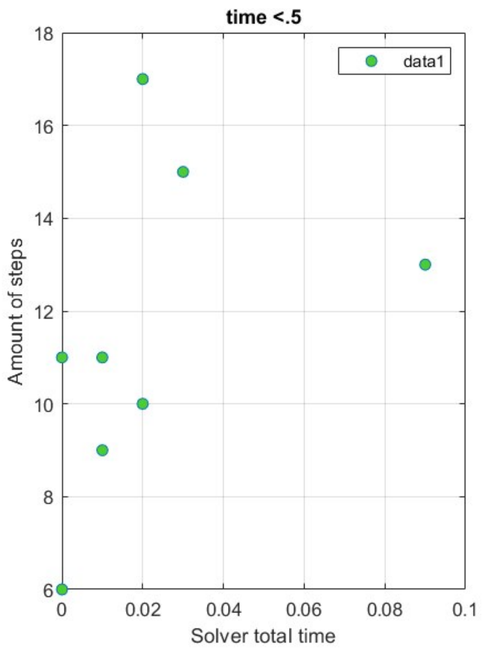
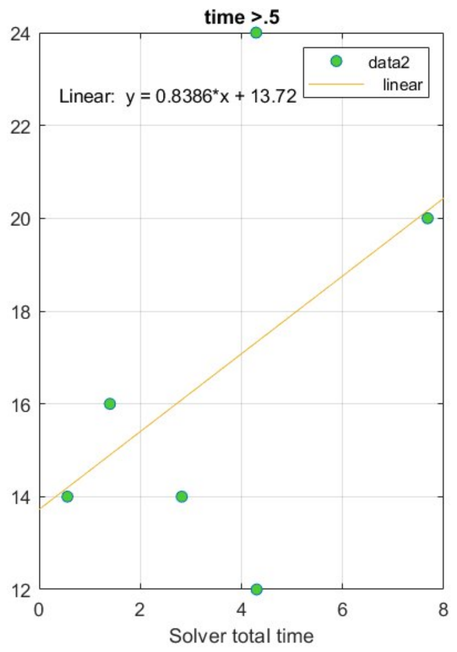


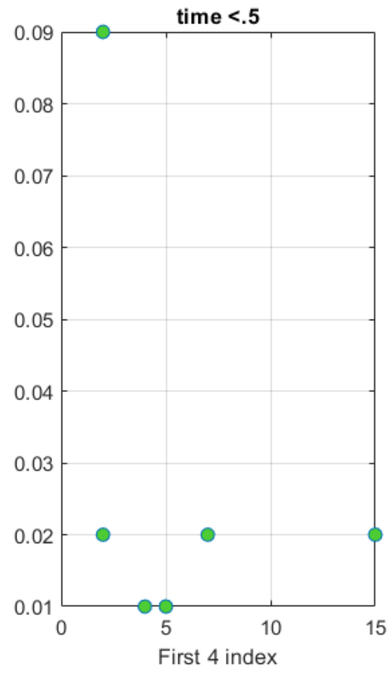
Table 1

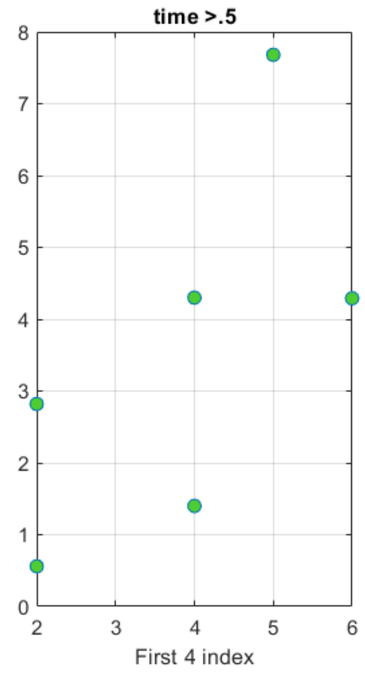
Parameter file	SolverNodes	SolverSolveTime	Savile Row TotalTime	Amount of Steps	Steps encoded in the order that was presented in the instructions
Item 1	1.00	0.00	0.21	6.00	[1, 2, 1, 2, 1, 2]
Item 2	17.00	0.00	0.52	9.00	[1, 1, 1, 2, 4, 5, 2, 5, 2]
Item 3	64.00	0.00	0.65	10.00	[1, 4, 1, 5, 2, 1, 4, 1, 5, 2]
Item 4	1.00	0.00	0.30	11.00	[1, 1, 2, 1, 2, 1, 1, 1, 2, 1, 2]
Item 5	2,383.00	0.09	0.86	13	[1, 4, 1, 5, 2, 4, 5, 2, 1, 4, 5, 5, 2]
Item 6	14,728.00	0.10	0.81	14.00	[1, 4, 5, 4, 5, 4, 5, 2, 4, 5, 2, 1, 5, 2]
Item 7	27.00	0.00	0.57	11.00	[1, 1, 1, 4, 6, 2, 1, 6, 2, 7, 2]
Item 8	10.00	0.00	0.84	15.00	[1, 2, 1, 1, 1, 1, 1, 2, 1, 1, 2, 1, 1, 1, 2]
Item 9	11.00	0.00	0.55	10.00	[1, 2, 1, 1, 2, 1, 4, 3, 1, 2]
Item 10	175,020.00	7.78	1.34	20.00	[1, 1, 1, 1, 4, 7, 7, 2, 5, 2, 3, 7, 7, 7, 2, 5, 2, 4, 5, 2]
Item 11	75,780.00	4.25	1.54	24.00	[1, 1, 1, 1, 2, 4, 7, 2, 2, 6, 2, 3, 2, 2, 6, 2, 3, 2, 6, 2, 3, 2, 6, 2]
Item 12	967,115.00	98.75	2.63	40.00	[1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 4, 3, 3, 3, 3, 3, 3, 3, 3, 7, 7, 7, 2, 8, 2, 8, 2, 6, 2, 7, 2, 8, 2, 6, 2, 3, 2, 6, 2, 2]
Item 13	80,800.00	2.79	1.26	14.00	[1, 4, 1, 1, 2, 5, 5, 4, 5, 2, 5, 5, 5, 2]
Item 14	49,044.00	1.37	0.79	16.00	[1, 1, 1, 4, 3, 5, 2, 4, 5, 5, 5, 5, 5, 2, 8, 2]
Item 15	136,583.00	4.28	0.84	12.00	[1, 1, 1, 4, 3, 6, 2, 7, 7, 2, 7, 2]
Item 16	11.00	0.00	0.93	17.00	[1, 2, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 2, 1, 4, 1, 2]
Item 17	0.00	0.00	0.69	0.00	[]
Item 18	1,353.00	0.08	0.96	0.00	[]
Item 19	0.00	0.00	0.16	0.00	[]
Item 20	5,590.00	0.33	1.05	0.00	[]


without item 12
Linear: y = 1.2*x + 12
Quadratic: y = - 0.12*x^2 + 2*x + 11
data1
linear
quadratic
Amount of steps
Solver total time


time <.5
data1
Amount of steps
Solver total time


time >.5
Linear: y = 0.8386*x + 13.72
data2
linear
Solver total time


time <.5
First 4 index


time >.5
First 4 index