Automated Warehouse Picking: A Multi-Agent Path-Finding Problem

Sami Hatoum

1 Overview

The automated warehouse problem is one instance from the set of multi-agent path-finding problems; these problems centre around the computation of collision-free paths for a group of agents from their starting locations to a target location. This specific problem defines a valid path as one which leads from a robot (the agent) to a target destination while 'visiting' an associated target object along the path, this must be achieved for all objects in an object list. Progressing beyond valid solutions, the larger objective of this problem scenario is the optimisation of total moves made by all robots on the grid. This implementation of the warehouse problem is achieved through the use of *Savile-row* running *Essence-prime* - a declarative language, in which the focus is on the logic of the computation, allowing the underlying system to determine the steps to reach a desired result. This report will discuss the design choices made selecting decision variables and building constraints, followed by empirical evaluations of different back-end solvers.

Assumptions

Certain assumptions have been made regarding any potential areas of ambiguity in the specification:

- **The inventory cannot be filled on the first time-step** - This being time-step 0; as this is not a robot's first action, it can only pick up a box starting from time-step 1
- **Robots are not forced to pick up the first box they pass over** - Depending on optimality, robots choose which box to pick up along their path
- **Boxes cannot begin on their own destination** - Boxes can begin on the destination of others as shown in p15, but not on their own
- **Destinations cannot appear above walls** - A solution would not be valid otherwise
- **Robot starting positions are provided in order of where they appear on the grid** - Assuming the top left of the grid is considered the earliest position and the bottom right as the latest
- **Robots cannot begin on the same grid cell** - This will be covered by the 'Part 2' constraints

2 Variables, Domains, and Constraints

2.1 Decision Variables and Constants

As per the practical specification, the solution to the automated warehouse problem comes in the form of a three-dimensional matrix, comprised of the coordinates every robot is occupying for each time-step in the total path. The path behaves as the primary decision variable, with others such as the inventory matrix, acting as auxiliary variables by which we are able to channel solutions to-and-from the path matrix.

Constants

- **NMAX** - Domain ranging from 1 to the maximum of the rows and columns of the grid. Used as the domain for the path as well as other variables relating which store coordinate values
- **TASKS** - Domain ranging form 1 to ntasks. Used to index all matrices pertaining to task completion, such as the inventory and task intervals
- **TASKS0** - Domain ranging form 0 to the total tasks. Used as the domain for the inventory. 0 to represent an empty inventory

- **INTERVAL** - Domain ranging from 1 to 3. Used to store intervals of task completion as three-tuples. INTERVAL-ROBOT, INTERVAL-PICKUP, and INTERVAL-DROPOFF are constants relating to three indexes in the tuple
- **ROBOTS** - Domain ranging from 1 to nrobots. Used to index all matrices pertaining to the actions of robots, such as the path, inventory, and robot moves.
- **STEPS** - Domain ranging from 0 to time-limit. All robots begin at time-step 0, as will be expanded on further, thus the indexing begins at 0
- **TOTAL-MOVES** - Domain ranging from 1 to the product of time-limit and nrobots. This upper bound would imply that on every time-step all robots have moved

Decision Variables

- **Path** - The primary decision variable, the domain of this matrix's time-steps ranges from 0 to the time-limit, a detail which is propagated across the decision variables concerning time. 0-indexing is utilised here to include the starting position of the robots, as opposed to beginning at the first move made by the robots
- **Inventory** - A two dimensional integer matrix used to store what box (if any) is being held by the robots at all time-steps. This will prevent the same box from being selected by the robots more than once to pickup as well as restrict all robots to a singular transportation operation at a time
- **Interval** - A two dimensional integer matrix which stores the essential information regarding a box's 'movement' from its pickup location to the where it is dropped off. This information is stored as a three-tuple for every task; the first element being the robot, and the remaining two correlating to the two time-steps indicating the start and end of the robot-box interaction
- **Robot-Task-Occurrence** - A two dimensional switch matrix used to formulate an implicit constraint regarding how many times the same task is completed between the robots
- **Robot-Moves** - A two dimensional boolean matrix which stores the difference in subsequent robot moves as a truth value; true if the robot has moved between time-steps, and false otherwise
- **Total-Moves** - A single integer, the summation of all moves by all robots. The program is aiming to minimise this integer, finding a solution in as few moves as possible. The domain of this value ranges from 1 to the maximum possible moves, that being a move by every robot at every time-step (reaching the time-limit)
- **Min-Moves** - A single integer, the lower bound on the total moves summation, the minimum moves made by a single robot in a solution

2.2 Parameter Constraints

Constraining the parameters proves quite trivial, introducing only a single challenge which was not inherently covered by the domains. For all parameters stored as coordinates, the robot starting positions and the task destinations, it had to be ensured they both lie in the bounds of the grid while only using a single domain in their declaration - the maximum of rows and columns. Thus, they are both iterated through, and their rows and columns are individually checked against their associated parameters.

2.3 Robot Constraints

Development of robot constraints came as part of an incremental process, first ensuring the robots at any timesteps fit within the boundaries of the grid and did not occupy the same tiles as walls. From there, subsequent moves were examined as to progress from random placements on the grid to a singular route. To ensure this constraint, the summation of the absolute difference of coordinate-pairs between a robot at its current time-step and the proceeding time-step should be 1 or 0. Respectively, these either indicate a movement in one of the four directions or no move at all.

```
$ Every subsequent timestep is at most one move away from the previous timestep
forAll robot : ROBOTS .
    forAll timestep : STEPS .
        (
            timestep < time_limit -> (
```

```
            (|path[robot, timestep, 1] - path[robot, timestep+1, 1]| +
             |path[robot, timestep, 2] - path[robot, timestep+1, 2]| = 1 /\
             robot_moves[robot, timestep] = true)
        \/
            (|path[robot, timestep, 1] - path[robot, timestep+1, 1]| +
             |path[robot, timestep, 2] - path[robot, timestep+1, 2]| = 0 /\
             robot_moves[robot, timestep] = false))
    ),
```

Listing 1: Figure 2.3.1: Robot Constraint - Subsequent Steps

Simultaneously, what is being recorded alongside this calculation is a truth value indicating whether that robot has moved or not, used in the later summations to find the total moves across the robots.

A detail of particular importance is the 0-indexing of the robot constraints, The 0th position in all robot paths equates to the starting positions of the robots, and behaves as a mapping to the robot positions in the parameter matrix **robot-pos**. This achieves an inherent form of symmetry breaking without the need to assign an explicit lexicographic indexing to the solution, detailed further in section 3.1: Symmetry Breaking.

The final robot constraint aims to formulate the bridge between the path and inventory of robots. This solution focuses on satisfying the actions of agents, not the agents themselves (each box has a valid interval as opposed to focusing on all robot paths). Thus, instead of iterating through all robots and time-steps, the universal quantifier is placed on the tasks. Shown in the logic below, for all tasks, there exists a robot, a start time-step, and an end time-step, such that the particular task is present on a path at some point in the solution. This singular constraint is dense and can be divided as such:

```
$ For all objects in the task list, there exists a pickup and drop-off time-step on a
    robot's path
forAll obj : TASKS .
    exists robot : ROBOTS .
        exists box_timestep, dest_timestep : STEPS .
            $ The drop-off occurs after the pickup
            box_timestep < dest_timestep
            /\
            (
                $ The robot has found a destination point for that box
                task_dest[obj, 1] = path[robot, dest_timestep, 1] /\
                task_dest[obj, 2] = path[robot, dest_timestep, 2]
            )
            /\
            map[path[robot, box_timestep, 1], path[robot, box_timestep, 2]] =
                task_obj[obj]]
            /\
            $ The object's interval is populated with the robot and both time-steps
            interval[obj, 1] = robot /\
            interval[obj, 2] = box_timestep /\
            interval[obj, 3] = dest_timestep,
```

Listing 2: Figure 2.3.2: Robot Constraint - Interval

The first conditional ensures the box drop-off occurs after the pickup. The next attempts to find a mapping between the destination time-step coordinates and those which appear in the **dest-timestep** matrix. This process is repeated between the pickup time-step and the task object found in **task-obj**. The final element of this constraint maps these three pieces of information to a single row of interval, for analysis later in the program.

This action-based approach, as opposed to a robot-based solution, answers two glaring concerns: "does every robot need to complete a task?" and "how can we ensure robots are not performing multiple operations at the same time?":

- **"Does every robot need to complete a task?"** - Addressing the simpler of these first, no. This question proves trivial when examining scenarios on larger grids, specifically ones in which a robot is closer to multiple boxes (and their destinations) then a secondary robot is to reaching either, thus amounting more moves in just the travel to reach the boxes. Implementing this would not only result in a less optimal solution, but the approach to determine whether or not a box should make the travel is in itself a flawed system (see section 3.4: Manhattan Distance).

- **"How can we ensure robots are not performing multiple operations at the same time?"** - This question poses a much more difficult challenge if iterating through robots; in this implementation of the warehouse problem, there would be no way to tell if a robot has already started an operation (picking up a box) before beginning another. This is because finding a box's pickup and drop-off time-steps would not be able to indicate what other box has been picked up prior to this interval. Here, it would need to be established whether or not a previously picked-up box has been placed down, which will require internal loops within the main equal to the number of previous box-destination pairs found. This would not only be highly inefficient, but also infeasible to program as this would seemingly require an imperative language, to determine how many times iteration would be necessary as the solution is being found.

Two final constraints exist to ensure all robots take valid paths, denoted as Part-2 of the specification, no two robots can occupy the same location, nor can any two robots swap places between subsequent time-steps. These constraints are conditionally checked, only if there is exists more than one robot.
Addressing the former condition, an **AllDiff** matrix comprehension is placed on the flattened interpretation of the path. Here, to produce a single continuous indexing of grid coordinates (as opposed to a 2D representation through rows and columns), the x-coordinate is taken and multiplied by the number of columns, then summed with the y-coordinate. No two values being equal at all time-steps guarantees no robots occupy the same grid cell.

```
$ No two robots occupy the same space on the grid
nrobots > 1 -> (
    forAll timestep : STEPS .
        allDiff([path[robot, timestep, 1]*ncols + path[robot, timestep, 2] | robot :
            ROBOTS])
),
```

Listing 3: Figure 2.3.3: Part 2 Constraint - Unique Robot Positions

The latter of these constraints is achieved by iterating through all robots and time-steps. Using the negation of an existential, there cannot exist two robots such that their coordinates are equal between subsequent time-steps. This condition must be checked both ways else this is not considered a swap.

```
$ A robot cannot swap places with a robot between subsequent timesteps
nrobots > 1 -> (
    forAll robot : ROBOTS .
        forAll timestep : STEPS .
            timestep < time_limit ->
            !(
                $ There doesn't exist a robot which takes the position of the current
                    robot
                $ in the subsequent time-step, this is checked both ways
                exists next_robot : int(robot+1..nrobots) .
                path[robot, timestep, 1] = path[next_robot, timestep+1, 1] /\
                path[robot, timestep, 2] = path[next_robot, timestep+1, 2] /\

                path[next_robot, timestep, 1] = path[robot, timestep+1, 1] /\
                path[next_robot, timestep, 2] = path[robot, timestep+1, 2]
            )
),
```

Listing 4: Figure 2.3.4: Part 2 Constraint - Robot Swapping Prevention

2.4 Inventory Constraint

Constraining the inventory behaves as a form of channelling between it and the path matrix, using the intervals as a communication line between the two - if the inventory is valid, the path must be valid between those time-steps where the inventory is full, and equally, if the path is valid between those time-steps, the inventory is too. Illustrated in figure 2.4.2, channelling is achieved through the mapping of indexes across matrices. With the set of intervals previously established, objects can be iterated through, and information of their interval is extracted. In figure 2.4.1, the time-steps are iterated through in the range [pickup, drop-off), ensuring the inventory is populated between these steps, and the object 'placed down' at the drop-off point. A successful pickup and drop-off also translates to a success within the **robot-task-occurrence** table (see section 3.3: Robot-Task Occurrence Tracking).

```
$ The inventory is set to that object's index from [pick-up, drop-off)
forAll obj : TASKS .
    (
        $ For all time-steps in the interval, the inventory is equal to the box
        forAll interval_step : STEPS .
            (
                interval_step >= interval[obj, INTERVAL_PICKUP] /\
                interval_step < interval[obj, INTERVAL_DROPOFF]
            ) ->
            inventory[interval[obj, INTERVAL_ROBOT], interval_step] = obj
    )
    /\
    $ At drop-off, the inventory is 0
    inventory[interval[obj, INTERVAL_ROBOT], interval[obj, INTERVAL_DROPOFF]] = 0
    /\
    $ A task has successfully been completed by a robot, marking it in the occurrence
        table
    robot_task_occurence[interval[obj, INTERVAL_ROBOT], obj] = 1,
```

Listing 5: Figure 2.4.1: Inventory Constraint - Interval

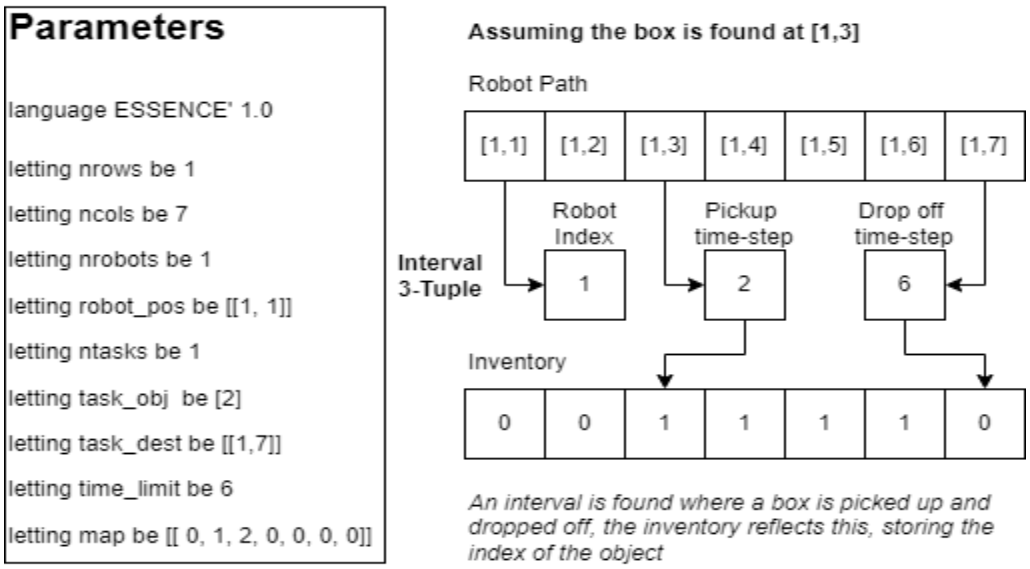

Figure 2.4.2: Inventory Example - Channelling

When deciding on the domain of the inventory matrix, fullness boolean indicators of the inventory did not suffice. This relates back to issue of robots not engaging in multiple tasks simultaneously; there would be no way to tell if two tasks are being done concurrently. A numeric system was thus adopted, this system relies on the indexes of tasks as opposed to the value of the box to combat the same issue in situations containing multiple boxes of the same colour. Shown in figure 2.4.3 below are examples detailing how these two unused models would fail - there is no valid path to the example though they will both produce a false positive solution.

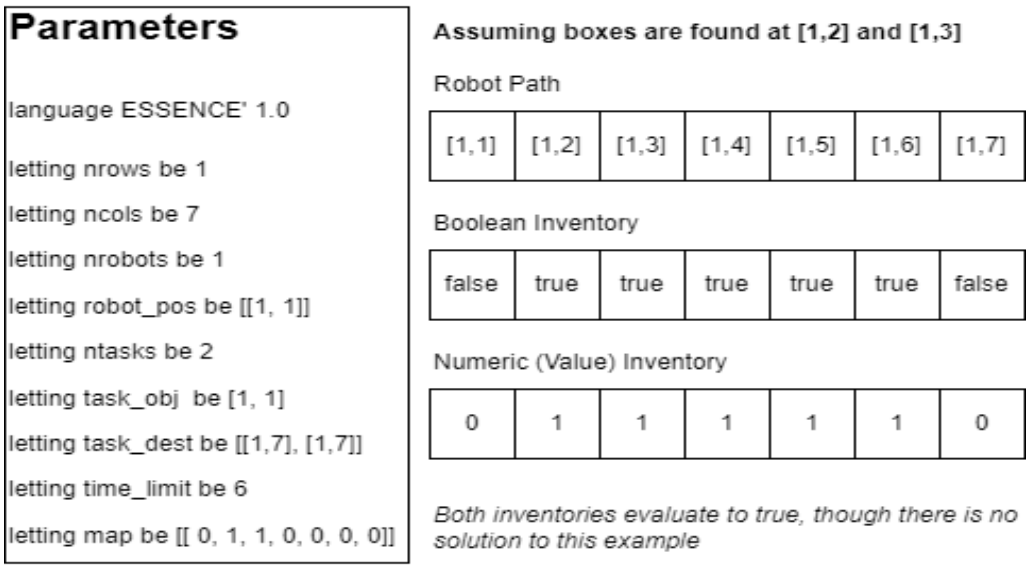

Figure 2.4.3: Inventory Example - Invalid Alternatives

2.5 Interval Constraint

One additional implied constraint placed on the intervals themselves is to ensure there exists no overlap between the intervals of different objects found on the same robot's path. Thus, for all distinct intervals which share the same robot, one's drop-off must occur before the other's pickup.

```
$$ INTERVAL $$

$ For all intervals of a robot, none can overlap
forAll obj1, obj2 : TASKS .
    (obj1 != obj2 /\ interval[obj1, INTERVAL_ROBOT] = interval[obj2, INTERVAL_ROBOT])
        ->
    (
        interval[obj1, INTERVAL_DROPOFF] < interval[obj2, INTERVAL_PICKUP] \/
        interval[obj2, INTERVAL_DROPOFF] < interval[obj1, INTERVAL_PICKUP]
    ),
```

Listing 6: Figure 2.5.1: Interval Constraint

2.6 Minimisation Constraints

Denoted as Part-3 of the specification, we attempt to minimise the sum of travel distances, thus maximising the total time all robots spend stationary. Covered in the robot constraints, this condition can be checked, and recorded, by finding the difference in subsequent time-steps for each robot.

The summation of this can be attained by finding all the truth values contained in the **total-moves** matrix. However, an additional implied constraint is placed on this final sum as well; the minimum total steps of the robots is at least as large as the robot which moved the least, multiplied by the number of robots.

```
$ Minimise the sum of travel distances of all robots
total_moves =
    sum robot : ROBOTS .
        sum timestep : STEPS .
            robot_moves[robot, timestep] = true,

$ The lower bound on the total moves is equal to the minimum moves taken by a robot *
    nrobots
min_moves =
    min([sum timestep : STEPS . robot_moves[robot, timestep] = true | robot : ROBOTS
       ]) * nrobots,
```

```
total_moves >= min_moves,
```

Listing 7: Figure 2.6.1: Minimisation Constraints

Additional constraints have been considered to optimise this calculation more, such as negating moves if they are the exact inverse of the previous, however this proves to be additional computation with no impact on the final solutions. Furthermore, this particular optimisation could only be done if the 'intentions' of the robot are known, such as if that inverted move was done to provide passage for another robot, which would be impossible to determine.

3 Additional Constraints

3.1 Symmetry Breaking

Within the path matrix, the domain on the time-steps range between zero and the time-limit. This lower bound is of particular importance as it inherently breaks the symmetry of the solution; prior to this, a hypothetical parameter configuration containing four robots and a single solution, could equally be expressed as $P(4,4) = 24$ separate solutions, where robots and their paths occupy different rows of the matrix, offering the same solution functionally, in a different order.

In order to prevent these mirrored solutions from ever being searched for, a fixed ordering can be placed on the path matrix, whereby its time-step indexing begins at zero, allowing a bijective mapping to be made from the the robots in path at time-step zero, to their equivalent index robot position found in **robot-pos**.

This indexing also breaks another form of symmetry; in example figure 3.1.1 below, assuming the first time-step is one move away from a robot's starting position, the two robots starting in an diagonal orientation, would have the same availability of choice when selecting their starting position (for 2 out of 5 clashing move options). Thus, there would exist two equal solutions depending on which "X" the robots start at; equivalent solutions can thus be cut from the search space earlier.

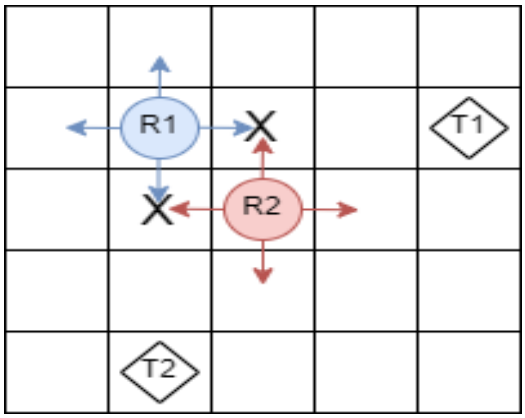

Figure 3.1.1: Symmetry Breaking Example - Clashing Start

3.2 Branching

As a further form of optimisation, branching specifies a sequence of variables for the solver to branch on, allowing for selective exploration of solution space. By strategically branching on certain solutions, we can prune large portions of the search tree without declaring explicit constraints to do so. In this solution, intervals were found to be the most optimal area to expand on as this matrix provides all the necessary information the solver would need to make educated decisions regarding short-listing, exploration, and backtracking.

3.3 Robot-Task Occurrence Tracking

An implied constraint, which aids to prune invalid solutions containing repeated tasks, comes in the form of an occurrence table. Here, rows are comprised of the robots and columns are the tasks present in **task-obj**. Taking the summation of each column, it is possible to restrict the total occurrences of a task's completion to a desired result, one. This constraint would eliminate any potential solutions in which the same task is repeated with no added benefit

to the problem.

Through extensive testing, this constraint only requires one side of its conditional to be explicitly stated; when a task is completed by a robot, a one is placed in the equivalent position in the matrix, however, incomplete tasks are represented as zeros, and are thus defaulted to by the solver (as they take the minimum value in the domain), requiring no constraints to implement.

3.4 Manhattan Distance

When considering the implementation of the Manhattan distance, the objective of this constraints was: "to calculate the minimum Manhattan distance for each robot to each object in the object list"; that is, to find the minimum distance for each robot to an object's pickup location in order to better determine, and order, the tasks a robot should prioritise.

Though we would want to apply this constraint to all scenarios, in practice this would be primarily a conditional constraint due to one striking limitation in this solution as a whole - How can we know which box of the same colour is chosen for the Manhattan distance? As there exists no bijective mapping from **task-obj** to the boxes present on the map, it would not be possible to know which one was selected for the distance calculation (since the box selected on the map is attained through an existential quantifier) leading to potentially pruned valid, and possibly optimal, solutions as well. Currently, a set of constraints to accommodate for these limitations are unknown, and whether these will slow down or speed up computation is also unknown.

4 Basic Empirical Evaluation

4.1 Evaluation Metrics

The evaluation of this solution to the automated warehouse problem is based on the results of 20 diverse parameter configurations, finding as many possible edge-case instances while simultaneously testing the limits of the grid size, robots, and boxes, before the solver cannot produce a solution (in a reasonable amount of time) before timing-out.

The basic empirical evaluation, will be an investigation against all parameter configurations using *minion* as the back-end solver, examining **SolverNodes**, **SolverSolveTime**, **Savile Row TotalTime**, and **Total Moves** and their implications. As many of the constraints are intertwined with one another, there exists no code in which we are able to isolate components of inherent symmetry breaking and implied constraints without dismantling the full solution. Thus the basic evaluation will be of the code in full, evaluating its performance against the primary back-end solver, minion.

We define a "reasonable amount of time" as no longer than 10 minute, the decision on this upper bound is entirely arbitrary, however as well be discussed, this limit proves unnecessary for most test cases.

4.2 Evaluation Results

The results shown below (table 1) indicate a clear correlation between the solver times - and nodes required - and the 'difficulty' of the problem. While this conclusion seems overt, the cause of difficulty can be expanded on further; the branching rate seems to grow exponentially based on one of three factors: grid size, robot count, and task count. Each of these three, once significantly increased, (in isolation or otherwise) are a distinct cause of parameters p13-p20 timing-out. For example, parameter p14, while seemingly trivial, with four boxes and four evident destinations, takes an exceedingly long amount of time to complete due to the large grid size, offering too many possible moves each robot could make.

Results indicate a clear cut-off point between instantaneous problem solutions and ones taking an indefinite (and unreasonable) amount of time; p11 seems to be this threshold, amounting to $\sim$5 minutes of solver time compared to the previous at $\sim$2 seconds. As the remaining parameter configurations only get more difficult it can be assumed none will be solved within a reasonable time frame, however p12 was also ran in order to extrapolate future expected wait periods; this took approximately 2 hours, far exceeding our upper time limit.

Perhaps one of the most important factors in the success of this model is control over the solver, through branching. Without it, running minion, the threshold is p5, with all subsequent parameter configurations running out of time. This single line of code drastically reduces the total search space, and heavily improves the model's performance.

Below are the results with branching on the intervals. Testing other auxiliary variables to branch on proves much less effective in reducing total solver nodes. This result aligns with our expectations exactly as branching over the intervals provides the solver with the most information to make educated search choices; the solver is effectively attempting to select solutions around valid intervals. Pairing this with constraints to limit each object to a single interval (the robot-task occurrence table), as well as constraints on the intervals themselves ensures we are narrowing down our total search nodes as much as possible.

Examining the effects of minimisation on the speed of the solver, there appears to be some deviation from initial expectations. At first, it was assumed that if we optimise the model, this would require more nodes and more time to solve than an unoptimised version for all instances. While for the majority of solutions this expectation is satisfied, certain configurations detail something different. Parameters, p5 and p10 are two examples of producing a slower solution though with also fewer nodes, when these two products were expected to be proportional. While the exact reason for this is unknown, a few reasons can be potentially inferred:

- **Hardware Differences** - Through some repeated testing, there exists minor differences in the time to solve depending on computer hardware, or running background software. These parameters performing slower can simply be due to slight delays in hardware, and with solver times generally below one second, these millisecond delays can prove significant
- **Early Stopping Criteria** - When the unoptimised solver is tasked with finding any solution, it can stop as soon as it finds a viable solution. This can cause the solver to avoid difficult calculations or pruning techniques that would otherwise slow it down in the pursuit of an optimal solution, though still allow for more nodes to be searched.
- **Potential Backtracking** - In instances where fewer nodes does not correlate to a faster solution, it could assumed that backtracking is being carried out on these few nodes to find an optimal solution amongst them

Table 1 and 2 illustrate these differences, with and without minimisation respectively. One slight change in the solutions can be seen in parameters p1 and p2, in which "minimisation (is) dropped". This is assumed to be due to only one existing solution/node, and thus nothing to minimise.

Instance	**SolverNodes**	**SolverSolveTime**	**Savile Row TotalTime**	**Total Moves**
example.param	3543	0.012515	0.768	9
p1	1	0.0201	0.769	3
p2	1	0.015719	0.699	6
p3	23	0	0.72	8
p4	4	0	0.937	5
p5	30	0	0.986	8
p6	2202798	7.18087	1.011	16
p7	688	0	0.774	10
p8	8649	0.023429	0.799	10
p9	170462	0.714783	1.325	23
p10	7494	0.041358	0.964	22
p11	48474981	336.874	1.489	32
Remaining test cases unable to solve within a reasonable time-frame				
p12	1008884989	7660.23(∼2 hours)	1.701	40
p13	N/A			
p14	N/A			
p15	N/A			
p16	N/A			
p17	N/A			
p18	N/A			
p19	N/A			
p20	N/A			

Table 1: Basic Empirical Evaluation with Minimisation

Instance	SolverNodes	SolverSolveTime	Savile Row TotalTime
example.param	37	0.027814	0.85
p1	1	0.019159	0.537
p2	1	0.020503	0.704
p3	19	0.016706	0.722
p4	4	0.020254	0.697
p5	53	0	0.783
p6	1072007	2.39847	1.05
p7	334	0	0.788
p8	7569	0.034439	0.816
p9	20936	0.18429	1.369
p10	74178	0.234154	1.09
Remaining test cases unable to solve within a reasonable time-frame			
p11	N/A		
p12	N/A		
p13	N/A		
p14	N/A		
p15	N/A		
p16	N/A		
p17	N/A		
p18	N/A		
p19	N/A		
p20	N/A		

Table 2: Basic Empirical Evaluation without Minimisation

5 Additional Empirical Evaluation

5.1 Evaluation Metrics

The additional empirical evaluation tests all parameter cases using *SAT* as the back-end solver, examining the **SolverTotalTime**, **SATClauses**, **SavileRowTotalTime**, **SATVars**, and **Total Moves**. The purpose of this evaluation is to ensure the total moves are consistent across different solvers, and whether or not new test cases can be solved by substituting the back-end.

5.2 Evaluation Results

The capabilities of SAT prove more successful compared to minion in its ability to tackle larger grid sizes, and more agents. Detailed below (table 2), within a reasonable time-frame, SAT is able to successfully complete the first 16 parameters (and the example) before 'timing out', and where SAT and Minion overlap, the total moves are consistent.

One distinction to be made here is the discrepancies in the inventory solutions. For reasons unknown, as the workings of the SAT solver are unknown, a filled inventory will interchange between indexes of the same box type. However, manually tracing the path each robot takes proves successful. Whether this be considered a valid solution or not is dependent on the importance placed on auxiliary variables rather than solely on the primary solution.

Instance	SolverTotalTime	SATClauses	SavileRowTotalTime	SATVars	Total Moves
example.param	0.03	3588	1.052	948	9
p1	0	0	0.187	0	3
p2	0	0.374	0	0	6
p3	0.02	1586	0.394	398	8
p4	0	115	0.3	36	5
p5	0	2549	0.396	603	8
p6	0.37	8935	1.046	1913	16
p7	0	3080	0.52	727	10
p8	0.05	6317	0.603	1341	10
p9	0.41	14555	1.155	3075	23
p10	0.06	9801	1.02	2085	22
p11	0.43	24440	1.594	5154	32
p12	0.3	29997	1.792	6169	40
p13	16.15	48500	3.515	10044	43
p14	1.34	49845	2.757	11492	41
p15	60.89	32416	2.042	6148	22
p16	1341.28	98618	4.718	19613	57
Remaining test cases done without minimisation, as they cannot be solved in time otherwise					
p17	0.58	151882	8.816	57798	N/A
p18	1.70	193205	10.832	153058	N/A
p19	3.27	235996	14.723	197065	N/A
p20	N/A				

Table 3: Additional Empirical Evaluation

6 Project Evaluation

This project proved to be a very interesting challenge, as this was my first experience with declarative programming; I enjoyed altering my standard approach to finding a solution to one in which all solutions are already found, and rather, we aim to find the correct one. This final solution underwent many iterations of different models and constraint approaches until what has been settled on, each with their own advantages and disadvantages outlined in the design section of the report. Ultimately, I am very satisfied with my model, if given more time I would attempt to implement additional constraints such as the Manhattan distance, and learn more about other back-end solvers, additionally, I would take averages of all tests for more precise solver times.

7 Conclusion

To conclude, the results of this practical aligned precisely with what I intended to achieve; this model is capable of producing a solution for the majority of, but not all, parameter configurations, as was expected when approaching the practical specification.