Modeling the Inglenook Shunting Puzzle

Helmut Simonis
Insight Research Ireland Centre for Data Analytics
School of Computer Science & IT, University College Cork, Cork, Ireland

Luis Quesada
Insight Research Ireland Centre for Data Analytics
School of Computer Science & IT, University College Cork, Cork, Ireland

Abstract

We present different models for the Inglenook shunting puzzle, a well known problem for model railway enthusiasts. The puzzle consists of rearranging wagons in a railyard with the help of a shunting locomotive to form a train with selected wagons in a specific order. This can also be seen as a deterministic planning problem, where the length of the plan to be generated is not known in advance. Traditionally, these puzzles are solved by hand, there is only limited literature analyzing the puzzle as a combinatorial problem. We present different models of the problem, and compare their efficiency on the different problem sizes. For the default problem size, all states (350k) and moves (2.2M) can be precomputed, the resulting directed graph can then be analyzed using Dijkstra's shortest path algorithm. We present an alternative view as a Constraint Satisfaction Problem modelled with large table constraints, which we test with different solver backends. Another alternative model uses string constraints to describe the allowed moves, without enumerating all states, we use the Z3 solver to search for solutions. A third model also does not rely on precomputed states, but models the states for a fixed-length path as constraints over finite domain variables. While there is an interest in solving this puzzle on its own, it also provides a scalable benchmark problem to test CP systems with large table constraints, string constraints, or complex logic formulas over finite domains.

2012 ACM Subject Classification Computing methodologies Planning and scheduling

Keywords and phrases Planning, Constraint Programming, Table constraint, Inglenook Puzzle

Acknowledgements This publication has emanated from research conducted with the financial support of Science Foundation Ireland under Grant number 12/RC/2289-P2, co-funded under the European Regional Development Fund. For the purpose of Open Access, the author has applied a CC BY public copyright licence to any Author Accepted Manuscript version arising from this submission. We are grateful to the reviewer who brought reference [10] to our attention.

1 Introduction

The Inglenook shunting puzzle is an example of a railway based planning problem. Given an initial state of wagons placed in the layout, the aim is to achieve a given target state in as few moves as possible. Each move is an operation involving a locomotive (also called shunter) moving wagons between locations of a limited capacity, coupling and uncoupling wagons as required. This is a very popular problem for railway enthusiasts, with many articles and videos describing how to build such a puzzle layout. Details of its history are for example given in [22]. The puzzle is typically solved by a human controlling the model shunter, setting the points, and performing the coupling or uncoupling of wagons. In this mode, the aim is to find a solution with the smallest number of moves. There are also two player competitions, where two identical layouts are operated by the players, the winner is determined by the shortest solution time, so that manual dexterity and familiarity with model railroad operations are also important.

The only scientific paper to discuss this specific problem we found is [7]. It considers under which conditions there are solutions to all puzzle instances, there is for example no

solution for every instance if we consider nine wagons in the traditional layout. The paper also gives (rather weak) upper bounds on the worst case optimal length of a solution for a given puzzle size, but does not discuss how to produce solutions for instances by a program.

There has been a steady stream of papers on train sorting and reversal [20, 5, 10, 1], but the underlying assumptions do not seem to match our problem. In practical terms, the problem of load shunting for railways has been largely replaced by the container pre-marshalling problem [9, 19].

In this paper we present three types of models for this problem, in addition to solving instances where possible by creating a graph of all legal states and movements, and finding solutions by a shortest path algorithm.

The first model encodes the possible states in domain variables for a path of a fixed length, with table constraints expressing the possible actions between states. As the resulting constraint graph is tree structured, and we can enforce domain consistency on the table constraints[14, 13, 23], we achieve global consistency of the problem by propagation alone [11], and find solutions without backtracking. But, given the large number of states and moves, the propagation requires significant resources, which leads to overall quite slow solution times.

The second model expresses the problem with constraints over strings of finite length. The actions of the shunter can be expressed as string concatenation, with additional integer constraints on the string length. While the model is quite compact, results using the Z3 solver are disappointing, perhaps due to our limited experience with this tool.

The third model expresses each state by a collection of domain variables which indicate for each slot of the layout whether the cell is empty, or contains one of the possible wagons. A global cardinality constraint [18] ensures that the correct number of values of each type is used in each state. The state transitions are expressed by a large disjunction of logical formulas describing each possible move. This formula can be concisely expressed as a MiniZinc [15] predicate. Different solvers perform very differently on this model, with clause learning solvers performing best.

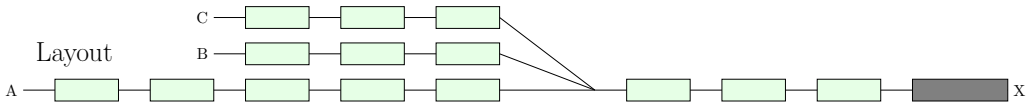
The paper is structured as follows: We are now at the end of the Introduction (Section 1). The next Section 2 describes the problem, and presents an example problem and its solution. We also discuss some of the properties of the default problem size by generating and analyzing all states and movements. In Section 3 we present three different models of the problem, a state-based CP model in Section 3.1, a model using finite string constraints in Section 3.2, and a position-based CP model in Section 3.3. We present experimental results in Section 4, before concluding with a summary and outlook in Section 5.

2 Problem Description

We now present the problem in more detail. Figure 1 shows the layout of the puzzle. On the left, we have three track areas, called A, B, C, which can hold 5, 3, and 3 wagons respectively. The slots available for wagons are marked as green rectangles. On the right we have track area X (also called head-shunt), which contains a locomotive (also called shunter) and space for three wagons. Tracks A, B, C are connected to area X by some points, which can be set for each move to connect X to either A, B, or C. The locomotive is coupled to any wagons in area X, and can either drop the left-most wagons from X into the selected area on the left, or pickup the right-most wagons in the selected area and pull those into the head-shunt. Each drop or pickup action can involve either one, two, or three wagons.

We explain the operation of the puzzle on a running example, given in Figure 2. We

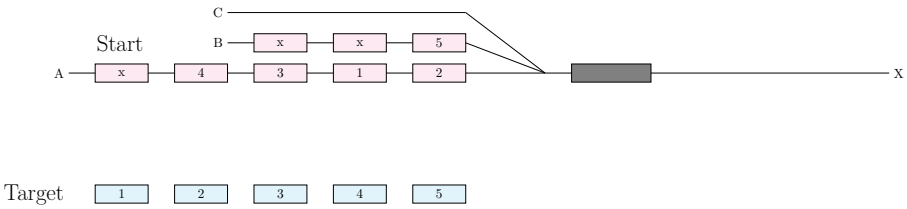
Figure 1 Layout of Puzzle



see the initial state at the top, and the target at the bottom. In the starting state, area A contains five wagons, area B contains three wagons, and areas C and X are empty (except for the shunter in area X). Five of the wagons are part of the target, they are numbered 1 to 5. The remaining three wagons are labeled x, we do not differentiate them, as they are not part of the target state. In the final state of the puzzle, area A should contain the wagons 1, 2, 3, 4, 5 in that order, we do not care where the other wagons are placed. There are problem variants which impose additional constraints on where the "don't care" wagons can be placed in the final state.

The fundamental challenge in the puzzle is that there is no random access to the wagons, each track area contains an ordered list of wagons, and all operations only concatenate and split these lists while maintaining the order within each list. To reverse the order of two wagons, we need to use multiple areas to temporarily place wagons, before reassembling the sequence in a different order.

Figure 2 Running Example

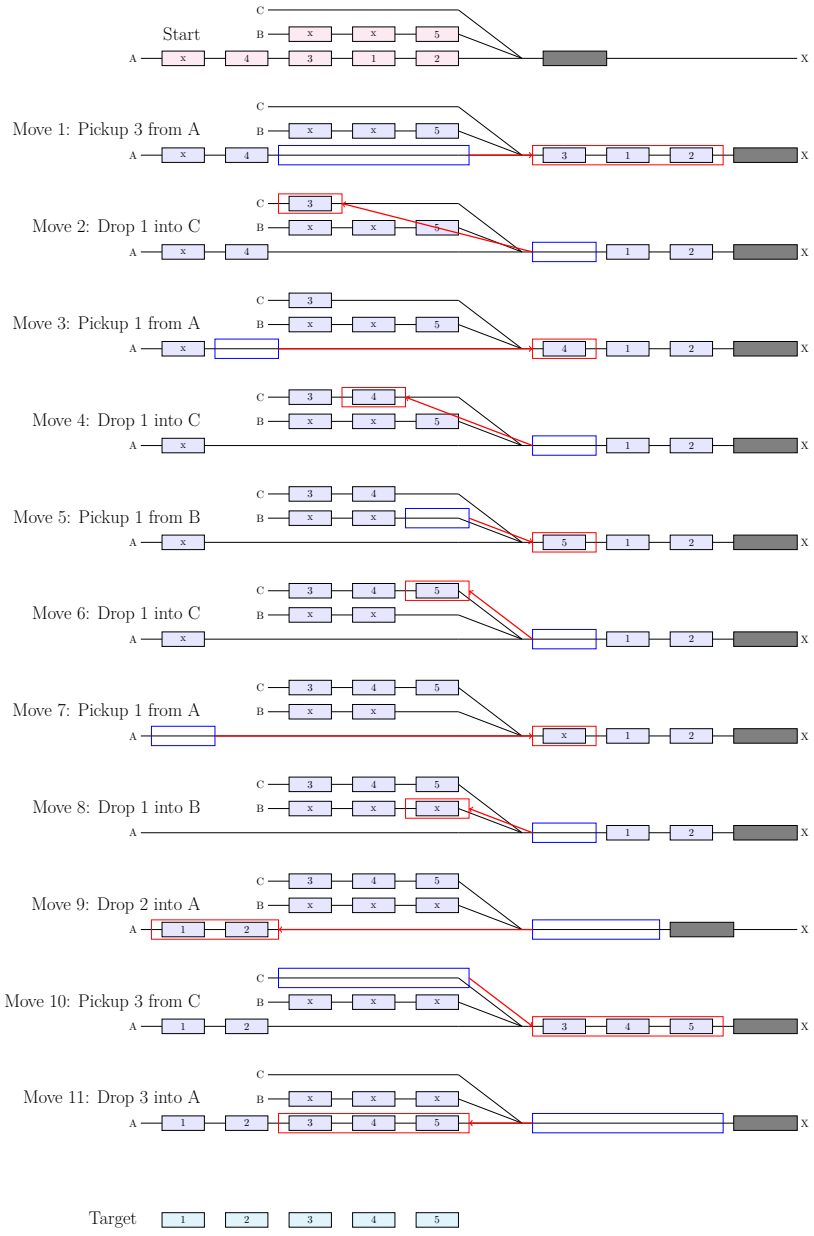


An optimal solution to the sample problem is shown in Figure 3, consisting of 11 moves. In the first move, we use the locomotive to pickup three wagons on the right end of area A, and move them to the head-shunt, area X. For every move, we see where the wagons moved originate (in blue), and where they end (in red). In the second move, we drop the left-most wagon in area X into the empty area C. Each move is either a pickup or a drop, involving one, two or three wagons. In principle, there are 18 possible actions, but in each state only a subset of the actions can be applied as areas are either already filled, or do not contain enough items to move.

In the last move, we drop the three wagons 3, 4, 5 from area X into area A, which already contains wagons 1, 2. This matches the target, and the puzzle is solved. The solution can be described as the sequence of actions performed. Note that there may be more than one optimal (i.e. shortest) solution for a given instance, while there are typically many more, non-optimal action sequences that achieve the target.

We can calculate the number of potential, legal states of the problem with the following observations. First, we can enumerate the possible distributions of wagons to the different track areas. We have eight wagons in total, and each area can hold between zero and three or five wagons. We find that there are 53 unique distributions. States belonging to different distributions are clearly different from each other. Within each distribution, we can rearrange the wagons freely, except that we do not distinguish between the three "don't care" wagons. There are $8!/3! = 6,720$ resulting permutations of ordering the eight wagons in each

Figure 3 Sample Solution



distribution. Two states based on the same distribution, but using different permutations will be different, so that there are $53 * 6,720 = 356,160$ distinct states for the default sized problem.

We can write a program that creates all states, and then checks which moves are possible between them. This leads to 2,284,800 potential moves. If a state contains five wagons in area A, and three wagons in area B, it is a valid starting state, a different instance of the puzzle. We find that there are 6,720 distinct puzzle instances. All states that contain the wagons 1..5 in increasing order in area A are valid end states, there are ten such states, considering that we do not differentiate the position of the three "don't care" wagons.

We construct a directed graph consisting of all states as nodes, and each legal move as a link between the origin and the target of the move. We add a dummy end-node which can be reached from each end state of the problem. We can then run a shortest path algorithm to determine the shortest path between a starting state and the end node. By reversing the orientation of the graph, we can run Dijkstra's algorithm to determine the distance of each node in the graph to the destination. This requires only a few seconds.

3 Models

In this section, we will introduce three different approaches to modeling the Inglenook shunting puzzle as a constraint problem. We follow the traditional modeling approach to solving deterministic planning problems with Constraint Programming [21, 6]: We assume an upper bound of the plan length, and encode each state as a collection of domain variables. Constraints are used to ensure that only legal states are generated, and that consecutive states are connected by legal moves. If we find a plan with the fixed length, we can then try to find a shorter plan by reducing the number of states allowed. If we do not find a solution, we have to allow a longer path, until we find a solution.

The first model assumes that we have already calculated all potential states and moves, and encodes each state as a single variable which ranges over all possible states, and the solution as a sequence of states that are linked by legal moves. The resulting model has many desirable properties, but is rather pointless, considering that we can just solve the shortest path problem on the implied state graph to find solutions. On the other hand, this model provides a very interesting benchmark for very large table constraints, showing a wide range of results from different solver implementations.

The second model uses string constraints [2][4][3] over finite strings to model the potential moves. Constraints over strings have been studied in a number of different constraint systems, but most works are experimental, and have not found their way into a ready-to-run system. We use the Z3 solver [8] which provides a finite string package.

The third model encodes states by a collection of variables, one for each slot for a wagon in the layout. The values are either zero, representing an empty slot, or the number of the wagon placed in the slot. Some constraints are used to force each state to be a legal state. A large disjunction links consecutive states by enforcing legal moves between states. The resulting model does not rely on any pre-computation, and expresses the problem in a very compact form.

3.1 State-Based Model

Our first model assumes that all states and moves are precalculated, and are represented in tabular form. The states are numbered from 0 to *target*-1, and the state transitions are available as a two dimensional integer array *moves*, where each row describes one move by

giving the source and the destination of the move. As described above, we add an overall end node with index *target*, and moves from all target states to this end node, and another move from the end node to itself. This allows to extend a solution by dummy moves at the end.

Program 4 gives the very compact MiniZinc model. We have an array of state variables x of length *size*, each state variable ranges from 0 to *target*. The first state is set to the required *start* state, the last state is the overall end state. We link all consecutive pairs of state variables with a *table* constraint using the *moves* array. We could add additional constraints, for example an *alldifferent_except_target* constraint to enforce that each state, except the end state, can only be used once.

Figure 4 MiniZinc Program for State-Based Model

```
include "globals.mzn";
int:start;
int:target;
int:size;
array[int,int] of int:moves;

array[1..size] of var 0..target:x;

constraint x[1] = start;
constraint x[size] = target;
constraint forall(i in 1..size-1)(table([x[i],x[i+1]],moves));

solve satisfy;
```

This program has many interesting properties. We can enforce domain consistency for each *table* constraint, different variants of that constraint have been developed over time. We also note that the constraint graph is a chain. We know that in this case propagation alone will enforce generalized arc-consistency, removing all inconsistent values from the domains of the variables. Each value left is part of a solution, so that we can find a solution by assigning values without backtracking. It also means that we detect inconsistency of a problem instance by propagation alone. If the selected path length is too small, propagation of the *table* constraints will detect that there is no solution before search is started. This is very helpful if we want to find the shortest solution length.

This model encodes all information about the states and legal moves into the table constraints, and achieves all possible propagation by enforcing domain consistency of each table constraint. As far as propagation is concerned, this is the best possible model, but it requires pre-computation of all states and moves, which may be too resource heavy for larger problem sizes, and will in any case need significant time to perform the propagation.

3.2 String-Based Model

This section explains our approach for solving the Inglenook Shunting Puzzle. The proposed method aims to find a sequence of moves that transforms the initial state into the target state by modeling the problem using string constraints. The approach is implemented using the Python binding for the Z3 SMT solver, which allows us to define constraints and transitions in a formal way. The Z3 solver is a powerful tool for solving logical formulas and is widely used in various applications, including formal verification and constraint satisfaction problems [8].

The problem is modeled using string constraints to represent the transitions between states. Each state is defined by four variables: A, B, C, and X, which represent the positions of the train cars and the shunting engine. The transitions between states are constrained by the following rules:

- **Capacity Constraints:** Each track has a maximum capacity:

$$\begin{aligned}&\text{Length}(A) \leq 5,\\ &\text{Length}(B) \leq 3,\\ &\text{Length}(C) \leq 3,\\ &\text{Length}(X) \leq 3.\end{aligned}$$

- **Transition Constraints:** The valid transitions are expressed as a disjunction of conjunctions:

$$\begin{aligned}&\big((A + X = A_{\text{next}} + X_{\text{next}}) \wedge (B = B_{\text{next}}) \wedge (C = C_{\text{next}}) \wedge (A \neq A_{\text{next}})\big) \vee\\ &\big((A = A_{\text{next}}) \wedge (B + X = B_{\text{next}} + X_{\text{next}}) \wedge (C = C_{\text{next}}) \wedge (B \neq B_{\text{next}})\big) \vee\\ &\big((A = A_{\text{next}}) \wedge (B = B_{\text{next}}) \wedge (C + X = C_{\text{next}} + X_{\text{next}}) \wedge (C \neq C_{\text{next}})\big).\end{aligned}$$

This formalization allows the Z3 SMT solver to compute valid sequences of moves that transform the initial state into the target state while respecting all constraints.

3.3 Position Based Model

Our third model also does not rely on pre-computation, and encodes each state as a collection of domain variables. For one state, we use one variable for each slot in the layout (see Figure 1). Theses variables can take one of seven possible values: The value zero indicates that the slot is empty, the values 1..5 indicate that one of the target wagons is placed in this cell, and the value 6 indicates that one of the "don't care" wagons is placed here. In our Program 5 we use arrays A, B, C, and X to hold theses variables for each state in our finite path-length model. We also introduce an array S, which combines the variables for each state into a single list. The constraints of the model either constrain the variables for each state to only allow legal states, or to link two consecutive states together. We first express the rows of the S array as the concatenation of the corresponding rows of the A, B, C, and X arrays. We then state that the first state must contain the wagons in the starting locations, and the last state must contain the wagons in the A array in the correct target sequence. The next constraint (Line 35) is a *global_cardinality* constraint, which ensures that each value for a state occurs the right number of times. For the variables of one state, the target values 1..5 occur exactly once, the "don't care" value occurs three times, and the empty value zero occurs six times.

The next constraints (Lines 39-43) enforce that the empty cells are at the beginning of arrays A, B, and C, and at the end of array X. This is an important step to remove unwanted symmetry, and to ensure that we can control how many non-zero elements are in an array by just checking a single cell.

We then use a predicate *move* to express the constraint of legal moves between any two consecutive states. This predicate is given in Program 6. Finally, there are four potential extra constraints (Lines 48-51) that play a dual role to the previous implications. If a cell is non-zero, then all cells to the right (resp. left for X) must also be non-zero. These constraints are not used in our default runs, they are only used in Table 9 as a variation of the model.

The definition of the *move* predicate in Program 6 links the variables of one state in arrays A, B, C, and X, with the variables of the next state in arrays $A1$, $B1$, $C1$, and $X1$. The definition consists of a large disjunction of all 18 potential moves, where each move is expressed by a logical conjunction in one line. As an example we consider the first entry (Line 9 in Figure 6), which describes the move "pickup one wagon from A into X". We can

Figure 5 MiniZinc Program for Position-Based Model

```
include "globals.mzn";

% constants, some read from command-line
int:size;
int:nrWagons;
int:lengthA=5;
int:lengthB=3;
int:lengthC=3;
int:lengthX=3;
int:lengthTotal = lengthA+lengthB+lengthC+lengthX;
int:nrZero = lengthTotal-nrWagons;
int:nrDontCare = nrWagons-5;

set of int:Run = 1..size;
set of int:Dom = 0..6;
array[int] of int:startA;
array[int] of int:startB;
array[int] of int:target=[1,2,3,4,5];

% variables
array[Run,1..lengthA] of var Dom:A;
array[Run,1..lengthB] of var Dom:B;
array[Run,1..lengthC] of var Dom:C;
array[Run,1..lengthX] of var Dom:X;
array[Run,1..lengthTotal] of var Dom:S;

% S as concatenation of A, B, C, and X for each state
constraint forall(i in 1..size)
    (row(S,i) = row(A,i)++row(B,i)++row(C,i)++row(X,i));
% initial and target conditions
constraint row(A,1) = startA;
constraint row(B,1) = startB;
constraint row(A,size) = target;
% global cardinality to enforce correct number of values in each state
constraint forall(i in 1..size)(global_cardinality(row(S,i),
    [0,1,2,3,4,5,6],
    [nrZero,1,1,1,1,1,nrDontCare]));
% implications to force zeros at start (resp. end for X) of shunt
constraint forall(i in 1..size,j in 2..lengthA)(A[i,j] = 0 -> A[i,j-1] = 0);
constraint forall(i in 1..size,j in 2..lengthB)(B[i,j] = 0 -> B[i,j-1] = 0);
constraint forall(i in 1..size,j in 2..lengthC)(C[i,j] = 0 -> C[i,j-1] = 0);
constraint forall(i in 1..size,j in 1..lengthX-1)
    (X[i,j] = 0 -> X[i,j+1] = 0);
% move constraint between two consecutive states
constraint forall(i in 1..size-1)(move(row(A,i),row(B,i),row(C,i),row(X,i),
    row(A,i+1),row(B,i+1),row(C,i+1),row(X,i+1)));
% potential extra constraints, not used in default mode
constraint forall(i in 1..size,j in 1..lengthA-1)(A[i,j] != 0 -> A[i,j+1] != 0);
constraint forall(i in 1..size,j in 1..lengthB-1)(B[i,j] != 0 -> B[i,j+1] != 0);
constraint forall(i in 1..size,j in 1..lengthC-1)(C[i,j] != 0 -> C[i,j+1] != 0);
constraint forall(i in 1..size,j in 2..lengthX)(X[i,j] != 0 -> X[i,j-1] != 0);

solve satisfy;
```

Figure 6 Transition Predicate for Position-Based Model

```
predicate move(array[1..lengthA] of var Dom:A,
               array[1..lengthB] of var Dom:B,
               array[1..lengthC] of var Dom:C,
               array[1..lengthX] of var Dom:X,
               array[1..lengthA] of var Dom:A1,
               array[1..lengthB] of var Dom:B1,
               array[1..lengthC] of var Dom:C1,
               array[1..lengthX] of var Dom:X1) =
(A[5]!=0 /\ X[3]=0 /\ A1=[0,A[1],A[2],A[3],A[4]] /\ X1=[A[5],X[1],X[2]] /\ B1=B /\ C1=C) \/
(A[4]!=0 /\ X[2]=0 /\ A1=[0,0,A[1],A[2],A[3]] /\ X1=[A[4],A[5],X[1]] /\ B1=B /\ C1=C)\/
(A[3]!=0 /\ X[1]=0 /\ A1=[0,0,0,A[1],A[2]] /\ X1=[A[3],A[4],A[5]] /\ B1=B /\ C1=C) \/
(A[1]=0 /\ X[1]!=0 /\ A1=[A[2],A[3],A[4],A[5],X[1]] /\ X1=[X[2],X[3],0] /\ B1=B /\ C1=C)\/
(A[2]=0 /\ X[2]!=0 /\ A1=[A[3],A[4],A[5],X[1],X[2]] /\ X1=[X[3],0,0] /\ B1=B /\ C1=C)\/
(A[3]=0 /\ X[3]!=0 /\ A1=[A[4],A[5],X[1],X[2],X[3]] /\ X1=[0,0,0] /\ B1=B /\ C1=C) \/

(B[3]!=0 /\ X[3]=0 /\ A1=A /\ B1=[0,B[1],B[2]] /\ X1=[B[3],X[1],X[2]] /\ C1=C) \/
(B[2]!=0 /\ X[2]=0 /\ A1=A /\ B1=[0,0,B[1]] /\ X1=[B[2],B[3],X[1]] /\ C1=C)\/
(B[1]!=0 /\ X[1]=0 /\ A1=A /\ B1=[0,0,0] /\ X1=[B[1],B[2],B[3]] /\ C1=C) \/
(B[1]=0 /\ X[1]!=0 /\ A1=A /\ B1=[B[2],B[3],X[1]] /\ X1=[X[2],X[3],0] /\ C1=C)\/
(B[2]=0 /\ X[2]!=0 /\ A1=A /\ B1=[B[3],X[1],X[2]] /\ X1=[X[3],0,0] /\ C1=C)\/
(B[3]=0 /\ X[3]!=0 /\ A1=A /\ B1=[X[1],X[2],X[3]] /\ X1=[0,0,0] /\ C1=C)\/

(C[3]!=0 /\ X[3]=0 /\ A1=A /\ B1=B /\ C1=[0,C[1],C[2]] /\ X1=[C[3],X[1],X[2]]) \/
(C[2]!=0 /\ X[2]=0 /\ A1=A /\ B1=B /\ C1=[0,0,C[1]] /\ X1=[C[2],C[3],X[1]])\/
(C[1]!=0 /\ X[1]=0 /\ A1=A /\ B1=B /\ C1=[0,0,0] /\ X1=[C[1],C[2],C[3]]) \/
(C[1]=0 /\ X[1]!=0 /\ A1=A /\ B1=B /\ C1=[C[2],C[3],X[1]] /\ X1=[X[2],X[3],0])\/
(C[2]=0 /\ X[2]!=0 /\ A1=A /\ B1=B /\ C1=[C[3],X[1],X[2]] /\ X1=[X[3],0,0])\/
(C[3]=0 /\ X[3]!=0 /\ A1=A /\ B1=B /\ C1=[X[1],X[2],X[3]] /\ X1=[0,0,0]);
```

apply the rule only if the last entry of A is not empty, and the last entry of X is empty. Remember that we enforce that all empty cells in A are on the left, and all empty cells of X are on the right. These two terms on the left form a guard which enforces the precondition of the rule to be usable. We then describe how the move of $A[5]$ shifts the elements of $A1$ to the right, creating a new empty space at the head of $A1$, and placing $A[5]$ at the head of array $X1$, removing the empty space at the end of X. In this move, arrays B and C are not changed, so $B1 = B$ and $C1 = C$. Note that the move enforces the invariant on the cardinality of each value occurrence. We remove one item from A, that is moved to $X1$, while we remove one empty cell from X, that re-appears to the left of $A1$.

Each line describes one potential move, but the moves can only be used when the guard conditions are enforced, so in reality only the moves feasible in a given states can be applied. The correctness of the program now obviously depends on the correctness of the *move* descriptions, which requires a careful check by hand.

Overall, the resulting program is not as compact as the state-based model, as each state is described by a collection of variables instead of a single variable. The resulting model also does no longer enforce domain consistency, so we rely on search to find feasible assignments. This creates a problem for back-end solvers which do not use clause learning, as their search method starts to thrash for larger problem instances. It makes it also very hard to rely on enumeration to show that there is no solution if the path length is chosen too small.

4 Experiments

In order to test the scalability of different solvers, we not only test the full-sized problem with eight wagons, but also studied problem instances with 5, 6, and 7 wagons. These problem variants are also sometimes used by humans wanting a slightly simpler puzzle instance. All of these instances use the default parameters (x/5/5/3/3/3/3). The parameters given are the total number of wagons, the number of wagons in the ordered solution, the number of spaces

in shunts A, B, C, and X, and the maximum number of wagons moved at any one time. We have also added problem sizes with 9 and 10 wagons, which significantly increase the number of states and moves, using parameters (9/5/5/4/4/4/3) and (10/5/5/5/4/4/3) respectively.

Table 1 shows the characteristics of the different problem sizes. For five wagons, both the number of states, as well as the number of moves is only a fraction of the full problem size of using eight wagons, these instances should be much easier to solve.

Table 1 Problem Instances

Wagons	Distributions	Permutations	States	Moves	Starting States	Finishing States	Min Moves	Max Moves
5	44	120	5,280	31,920	120	1	3	9
6	53	720	38,160	244,890	720	3	4	9
7	56	2,520	141,120	917,280	2,520	6	5	9
8	53	6,720	356,160	2,284,800	6,720	10	4	9
9	95	15,120	1,436,400	11,823,840	15,120	15	6	12
10	105	30,240	3,175,200	26,429,760	30,240	19	6	12

For the experiments, we ran the state-based and position based models on a Windows 11 laptop with a 2.3GHz Intel(R) Core(TM) i7-10875H CPU with 64 GB of memory. We used MiniZinc 2.93[15] from the command line, as the large data files created issues with the IDE version. We used the provided back-end solvers for Chuffed, CP-Sat [16], and Gecode [12], but also tested stand-alone solver back-ends for IBM's CPO (version 22.1.0), Choco-solver [17] (version 4.10.18) and Jacop (version 4.7.0) in Java. We used Cplex (version 22.1.0) only for the position-based model, as results for the state-based model were disappointing, possibly due to the mapping of the table constraints to linear constraints in the MiniZinc compiler. Where possible, we use 8 worker threads for the solvers. The Z3 [8] experiments were conducted on a MacBook Pro (2021) equipped with an Apple M1 Pro chip and 32 GB of memory. The machine runs macOS Sequoia 15.5 and utilises Python 3.11.11 alongside Z3 version 4.15.0.

For each problem size, we selected instances that require different shortest path lengths, using the precomputed states and moves. In the experiments below, we use one instance for each required, non-trivial path-length.

4.1 Results for State-Based Model

Table 2 shows the results of the state-based model on problem instances with six wagons, the largest problem size for which all solvers provided results. All solvers find solutions for all instances, with Gecode and Jacop struggling with the large instances. Note that all solvers find the first solution without backtracking, and time differences are due to either the propagation of the table constraints themselves, or to the scheduling of the propagation of multiple table constraints in a chain. The results for Chuffed, CP-Sat and CPO are very similar, as there is no search there is no advantage in using multiple threads for the enumeration phase for CP-Sat and CPO, compared to the single-worker search for Chuffed.

Table 3 shows the results for the eight wagon instances. Both Choco and Gecode[1] are not able to deal with the table constraints requiring over 2 million tuples. Jacop increasingly

[1] We tested a C++ standalone version of the Gecode model to check that the problem is in the library itself, not in the MiniZinc interface.

Table 2 State Model - Time (in Seconds) to Find First Solution for 6 Wagons

			MiniZinc				Java	
Config	Nr Moves	Nr Sols	Chuffed	CP-Sat	Gecode	CPO	Choco	Jacop
5 1234x	3	1	1.74	1.70	5.44	1.22	2.99	**0.52**
5 123x4	4	2	1.93	1.83	6.13	1.26	2.54	**0.52**
x 12435	5	4	2.32	2.14	7.70	1.37	3.11	**0.74**
5 124x3	6	3	2.52	2.27	8.78	**1.50**	4.15	1.60
x 14235	7	2	2.71	2.40	9.84	**1.51**	3.55	4.46
x 31245	8	2	3.15	2.73	11.14	**1.58**	4.02	16.01
x 21345	9	14	3.37	2.82	12.53	**1.68**	4.98	38.86
x 32145	10	16	3.56	2.85	13.23	**1.89**	5.98	62.75
x 21354	11	70	3.68	3.29	14.35	**2.43**	7.37	60.64
1 x5432	12	69	4.19	3.70	16.14	**2.02**	9.10	65.03

struggles to perform the propagation with the length of the solution path, indicating room for improvement either in the table constraint itself, or, more likely, in the constraint scheduling inside the solver. Again, results for Chuffed, CP-Sat and CPO are quite similar, this seems to indicate the current state-of-the-art for CP Solvers on this problem. The time taken, nearly a minute for the largest instances, shows the limits of the state-based approach. While performing only propagation, and zero backtracks, it takes a long time to achieve generalized domain consistency.

Table 3 State Model - Time (in Seconds) to Find First Solution for 8 Wagons

			MiniZinc				Java	
Config	Nr Moves	Nr Sols	Chuffed	CP-Sat	Gecode	CPO	Choco	Jacop
x5x 1234x	3	1	23.85	19.68	-	14.92	-	**3.81**
5xx 1234x	4	1	22.21	19.51	-	16.95	-	**5.76**
xxx 12435	5	2	26.55	24.12	-	19.45	-	**9.10**
3xx 12x45	6	1	28.35	25.14	-	20.47	-	**17.93**
xxx 14235	7	1	29.55	**25.02**	-	25.19	-	46.67
5xx 1243x	8	4	35.77	29.39	-	**24.39**	-	141.71
xxx 21345	9	4	40.18	30.40	-	**26.78**	-	474.84
xxx 41235	10	4	37.44	31.70	-	**29.93**	-	1,464.93
xxx 32145	11	4	43.28	36.96	-	**28.99**	-	1,781.49
xxx 34125	12	2	47.90	36.51	-	**30.91**	-	2,220.74
5xx x4231	13	5	44.99	37.06	-	**33.13**	-	3,196.63
5xx x3241	14	5	50.37	38.51	-	**35.36**	-	3,948.59
2xx 4351x	15	14	53.01	43.63	-	**36.46**	-	4,070.72
1xx x3254	16	16	55.97	45.67	-	**39.70**	-	5,296.88
21x xx543	17	33	60.52	47.81	-	**42.05**	-	6,456.42

Table 4 shows the results for instances with 10 wagons. Chuffed, CP-Sat and CPO are finding solutions within a timeout of one hour, spending up to 15 minutes on the initial propagation, but then requiring no search to extract all solutions. For this problem size, CP-Sat outperforms the other solvers. The results for the other problem sizes are given in

appendix B.

Table 4 State Model - Time (in seconds) to Find First Solution for Size 10 (Parameters 10/5/5/5/4/4/3)

Config	Nr Moves	Nr Sols	Chuffed	CP-Sat	Gecode	CPO	Choco	Jacop
xx5xx 1234x	3	1	230.11	220.99	-	254.91	-	**62.33**
xxx5x 123x4	4	2	255.93	226.85	-	273.20	-	**84.37**
xxxxx 12435	5	2	332.41	280.73	-	326.64	-	**142.80**
xxxxx 13245	6	6	339.16	**285.57**	-	399.61	-	560.39
xxxxx 41235	7	1	377.29	**292.67**	-	437.96	-	3549.13
xxxxx 31245	8	3	449.27	**341.49**	-	414.79	-	TO
xxxxx 21345	9	10	486.66	**343.58**	-	456.89	-	TO
xxxxx 43215	10	24	479.84	**357.94**	-	521.55	-	TO
xxxxx 21543	11	34	561.30	**412.78**	-	618.92	-	TO
5xxxx x2341	12	154	624.45	**422.22**	-	672.77	-	TO
5xxxx 32x41	13	128	608.73	**424.66**	-	651.12	-	TO
2xxxx 5314x	14	26	695.48	**448.25**	-	704.93	-	TO
1xxxx 543x2	15	73	740.45	**492.48**	-	823.29	-	TO
2154x xxx3x	16	77	731.95	**530.05**	-	878.32	-	TO

4.2 Results for the String-Based Model

Table 5 shows the results for the string-based model on problem instances with six wagons. The Z3 solver finds solutions for all selected instances, but solution times are quite variable, and, for the larger instances, significantly worse than the solution times needed by the state-based models.

Table 5 String Based-Model - Time (in seconds) and Number of Choices with Z3 for 6 Wagons

Config	Nr Moves	Time	Nr Decisions
5 1234x	3	0.29	921
5 123x4	4	0.35	2,764
x 12435	5	0.28	2,567
5 124x3	6	0.81	10,459
x 14235	7	1.06	9,086
x 31245	8	2.49	12,878
x 21345	9	4.21	25,282
x 32145	10	19.03	51,334
x 21354	11	210.39	107,414
1 x5432	12	347.98	184,332

Table 6 shows the solutions found for instances with eight wagons, where solutions are only found for path length less than 12. This is signiifcantly worse than the results for the state-based model (when the back-end solver are able to handle the instance size). Results for other problem sizes are found in Appendix C.

Table 6 String Based-Model - Time (in seconds) and Number of Choices with Z3 for 8 Wagons

Config	Nr Moves	Time	Nr Decisions
x5x 1234x	3	0.26	1,155
5xx 1234x	4	0.50	7,189
xxx 12435	5	0.47	5,810
3xx 12x45	6	54.91	73,402
xxx 14235	7	1.69	16,755
5xx 1243x	8	337.83	165,057
xxx 21345	9	7.87	39,837
xxx 41235	10	11.99	57,080
xx5 x4312	11	111.42	121,153
xxx 34125	12	TO	-
5xx x4231	13	TO	-
5xx x3241	14	TO	-
2xx 4351x	15	TO	-
1xx x3254	16	TO	-
21x xx543	17	TO	-

4.3 Results for Position-Based Model

Table 7 shows the results for the position based model from Section 3.3 for four different back-end solvers in MiniZinc on the eight wagon problem instances. The differences in the results for the different back-end solvers are striking. While Gecode can only handle the smallest path lengths, and Cplex also times out for large instances, both Chuffed and CP-Sat find solutions for all problem instances. It is not clear if the significant advantage of CP-Sat in execution time is only due to the multi-threaded search (using eight threads), compared to the single thread execution of the Chuffed backend, or if the MiniZinc compilation for CP-Sat produces a better model. CP-Sat execution with a single thread is often disappointing, and is not reported here (details in Table 21 in the appendix).

Table 8 shows the results for the position based model for instances with 10 wagons. Only Chuffed and CP-Sat are able to solve all instances within a 300 second timeout, with CP-Sat again providing much more consistent results, even for instances with more required moves. Results for the other problem instances are found in appendix D.

Overall, the position-based model using CP-Sat performs very well in finding a first solution, but it is not able to check infeasibility of larger problem instances. While the solution provided can be used to find solutions quickly, and those solutions are often optimal, at the moment we cannot prove optimality without creating the full state graph.

The implications and the global cardinality constraints of the position based model are redundant, as they are implied by the move constraints. We performed some experiments where we modify the MiniZinc program to see if these redundant constraints have a positive impact. The first variant adds the additional implications (Lines 48-51 in Figure 5) to the model, the second version is our default, used in all other reported results. The third variation removes the implications for zero values (Lines 39-43), while the next variant removes the global cardinality constraints (Line 35). The final variant only keeps the move constraints, more details are given in Appendix F. When we compare results for CP-Sat (Table 9) for size 8, we see that the global cardinality constraint is essential, without it even CP-Sat struggles to find solutions for the instances with larger move numbers, while the implications have a

Table 7 Position Model - Time (in seconds) to Find First Solution for 8 Wagons, All Solvers in MiniZinc

Config	Nr Moves	Chuffed		CP-Sat		Cplex		Gecode	
		Time	Nodes	Time	Failures	Time	Nodes	Time	Nodes
x5x 1234x	3	**0.30**	285	0.37	0	0.63	0	0.35	807
5xx 1234x	4	**0.33**	894	0.39	0	0.54	0	28.27	13,648,985
xxx 12435	5	**0.36**	2,970	0.50	203	0.89	0	92.77	39,097,442
3xx 12x45	6	0.56	10,897	**0.54**	261	1.11	0	TO	>120,581,535
xxx 14235	7	0.64	14,018	**0.61**	456	3.02	2,795	TO	>108,711,063
5xx 1243x	8	**0.48**	7,117	0.69	1,158	24.45	14,261	TO	>108,768,673
xxx 21345	9	1.10	26,053	**0.78**	2,414	91.66	30,611	TO	>81,069,139
xxx 41235	10	3.12	74,281	**0.94**	2,433	TO	>82,632	TO	>75,116,733
xxx 32145	11	3.47	87,001	**1.07**	3,233	TO	>78,093	TO	>94,125,683
xxx 34125	12	3.05	77,928	**1.34**	9,538	TO	>75,077	TO	>76,713,584
5xx x4231	13	5.35	124,490	**1.31**	8,116	TO	>62,390	TO	>65,023,177
5xx x3241	14	37.62	409,555	**2.81**	25,034	TO	>63,241	TO	>46,175,216
2xx 4351x	15	34.15	444,126	**2.72**	23,731	TO	>67,384	TO	>59,826,122
1xx x3254	16	15.67	287,364	**2.63**	20,361	TO	>50,273	TO	>42,507,680
21x xx543	17	35.28	421,762	**3.41**	33,689	TO	>60,772	TO	>43,260,349

Table 8 Position Model - Time (in seconds) to Find First Solution for 10 Wagons, All Solvers in MiniZinc

Config	Nr Moves	Chuffed		CP-Sat		Cplex		Gecode	
		Time	Nodes	Time	Failures	Time	Nodes	Time	Nodes
xx5xx 1234x	3	**0.36**	247	0.45	0	0.53	0	0.89	1,673
xxx5x 123x4	4	**0.37**	481	0.49	0	0.78	0	TO	-
xxxxx 12435	5	**0.43**	2,091	0.60	167	1.18	0	TO	-
xxxxx 13245	6	**0.55**	7,709	0.69	390	1.21	0	TO	-
xxxxx 41235	7	**0.75**	15,256	0.79	297	7.53	3,030	TO	-
xxxxx 31245	8	**0.88**	15,238	0.96	1,176	69.88	19,566	TO	-
xxxxx 21345	9	1.29	29,714	**1.00**	1,090	9.70	4,324	TO	-
xxxxx 43215	10	3.06	74,012	**1.15**	872	TO	-	TO	-
xxxxx 21543	11	2.25	58,035	**1.42**	4,874	TO	-	TO	-
5xxxx x2341	12	14.17	190,670	**1.78**	9,737	TO	-	TO	-
5xxxx 32x41	13	7.45	155,753	**2.10**	13,597	TO	-	TO	-
2xxxx 5314x	14	186.76	1,215,177	**3.73**	25,152	TO	-	TO	-
1xxxx 543x2	15	152.52	1,184,709	**4.02**	27,502	TO	-	TO	-
2154x xxx3x	16	107.89	758,501	**6.77**	53,847	TO	-	TO	-

smaller impact. This may also explain the relatively poor performance of the string based model, the Z3 solver currently does not seem to have constraints to constrain the number of occurrences of literals in the string variables.

Table 9 Position Model - Time (in seconds) to Find First Solution for Model Variants for 8 Wagons, CP-Sat Only

Config	Nr Moves	+Imply	Default	-Imply	-GCC	Move Only
x5x 1234x	3	0.34	0.37	0.37	0.98	**0.32**
5xx 1234x	4	0.39	0.39	0.37	**0.36**	0.37
xxx 12435	5	0.45	0.50	0.45	0.42	**0.37**
3xx 12x45	6	0.53	0.54	0.52	0.46	**0.45**
xxx 14235	7	0.60	0.61	0.59	0.61	**0.56**
5xx 1243x	8	0.67	0.69	0.62	**0.55**	0.80
xxx 21345	9	**0.71**	0.78	0.75	1.19	1.97
xxx 41235	10	0.98	0.94	**0.79**	2.20	2.99
xxx 32145	11	1.07	1.07	**0.92**	5.03	7.25
xxx 34125	12	**1.09**	1.34	1.23	13.65	13.11
5xx x4231	13	**1.14**	1.31	2.23	10.21	13.17
5xx x3241	14	**2.32**	2.81	3.02	50.15	61.70
2xx 4351x	15	2.49	2.72	**1.93**	30.68	171.75
1xx x3254	16	3.66	**2.63**	2.91	199.31	TO
21x xx543	17	**2.92**	3.41	3.91	262.22	54.82

5 Conclusions

In this paper we have explored different models for the Inglenook shunting puzzle, which is a popular problem solved by hand by model railway enthusiasts, but which can also be seen as an example of a deterministic planning problem. Our first model, which achieves generalized domain consistency, requires the pre-computation of all possible states and moves, and needs rather large table constraints. While this model is very simple, it offers little advantage over a shortest-path algorithm run on the implied state graph. The second model uses string constraints with finite strings in the Z3 solver. Results are preliminary, and would benefit from validation with another string solver. The third model, based on representing states as a collection of position variables, achieves very good results with CP-Sat (and, to a lesser extend, Chuffed), outperforming the state-based model by an order of magnitude. If we take the time required to find the state graph into account, it also outperforms the shortest-path approach which works on the complete state graph.

In this paper we only rely on the default search methods of the different solvers. But we can for example easily assign a heuristic evaluation to each move, based on the improvement in the problem difficulty it provides. Combining a (learned) heuristic with the constraint models studied could possibly result in an even faster solution to the problem.

The resulting problem instances show wide performance differences for some of the solvers used. They therefore might also be useful as a benchmark sets to compare table constraint methods, string solvers, or the use of complex logical formulas in a large disjunction of cases.

References

1 Jan-Alexander Adlbrecht, Benno Hüttler, Jan Zazgornik, and Manfred Gronalt. The train marshalling by a single shunting engine problem. *Transportation Research Part C: Emerging Technologies*, 58:56–72, 2015. URL: https://www.sciencedirect.com/science/article/pii/S0968090X15002466, doi:https://doi.org/10.1016/j.trc.2015.07.006.

2 Roberto Amadini. A survey on string constraint solving. *ACM Comput. Surv.*, 55(2):16:1–16:38, 2023. doi:10.1145/3484198.

3 Roberto Amadini, Pierre Flener, Justin Pearson, Joseph D. Scott, Peter J. Stuckey, and Guido Tack. Minizinc with strings. In Manuel V. Hermenegildo and Pedro López-García, editors, *Logic-Based Program Synthesis and Transformation - 26th International Symposium, LOPSTR 2016, Edinburgh, UK, September 6-8, 2016, Revised Selected Papers*, volume 10184 of *Lecture Notes in Computer Science*, pages 59–75. Springer, 2016. doi:10.1007/978-3-319-63139-4_4.

4 Roberto Amadini, Graeme Gange, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. String constraint solving: Past, present and future. In Giuseppe De Giacomo, Alejandro Catalá, Bistra Dilkina, Michela Milano, Senén Barro, Alberto Bugarín, and Jérôme Lang, editors, *ECAI 2020 - 24th European Conference on Artificial Intelligence, 29 August-8 September 2020, Santiago de Compostela, Spain, August 29 - September 8, 2020 - Including 10th Conference on Prestigious Applications of Artificial Intelligence (PAIS 2020)*, volume 325 of *Frontiers in Artificial Intelligence and Applications*, pages 2875–2876. IOS Press, 2020. doi:10.3233/FAIA200431.

5 Nancy Amato, Manuel Blum, Sandra Irani, and Ronitt Rubinfeld. Reversing trains: A turn of the century sorting problem. *Journal of Algorithms*, 10(3):413–428, 1989. URL: https://www.sciencedirect.com/science/article/pii/0196677489900370, doi:https://doi.org/10.1016/0196-6774(89)90037-0.

6 Philippe Baptiste, Philippe Laborie, Claude Le Pape, and Wim Nuijten. Constraint-based scheduling and planning. In Francesca Rossi, Peter van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*, pages 761–799. Elsevier, 2006. doi:10.1016/S1574-6526(06)80026-X.

7 Simon R. Blackburn. Inglenook shunting puzzles. *Electron. J. Comb.*, 26(2):2, 2019. doi:10.37236/8297.

8 Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008. doi:10.1007/978-3-540-78800-3_24.

9 Christopher Expósito-Izquierdo, Belén Melián-Batista, and Marcos Moreno-Vega. Premarshalling problem: Heuristic solution method and instances generator. *Expert Systems with Applications*, 39(9):8337–8349, 2012. URL: https://www.sciencedirect.com/science/article/pii/S0957417412002151, doi:https://doi.org/10.1016/j.eswa.2012.01.187.

10 Stefan Felsner and Martin Pergel. The complexity of sorting with networks of stacks and queues. In Dan Halperin and Kurt Mehlhorn, editors, *Algorithms - ESA 2008*, pages 417–429, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

11 Eugene C. Freuder. A sufficient condition for backtrack-free search. *J. ACM*, 29(1):24–32, 1982. doi:10.1145/322290.322292.

12 Gecode Team. Gecode: Generic constraint development environment, 2006. Available from http://www.gecode.org.

13 Christophe Lecoutre, Chavalit Likitvivatanavong, and Roland H. C. Yap. STR3: A path-optimal filtering algorithm for table constraints. *Artif. Intell.*, 220:1–27, 2015. URL: https://doi.org/10.1016/j.artint.2014.12.002, doi:10.1016/J.ARTINT.2014.12.002.

14 Christophe Lecoutre and Radoslaw Szymanek. Generalized arc consistency for positive table constraints. In Frédéric Benhamou, editor, *Principles and Practice of Constraint Programming - CP 2006, 12th International Conference, CP 2006, Nantes, France, September 25-29, 2006, Proceedings*, volume 4204 of *Lecture Notes in Computer Science*, pages 284–298. Springer, 2006. doi:10.1007/11889205_22.

15 Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. Minizinc: Towards a standard cp modelling language. In Christian Bessière, editor, *Principles and Practice of Constraint Programming – CP 2007*, pages 529–543, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

16 Laurent Perron and Frédéric Didier. Cp-sat. URL: https://developers.google.com/optimization/cp/cp_solver/.

17 Charles Prud'homme and Jean-Guillaume Fages. Choco-solver: A java library for constraint programming. *Journal of Open Source Software*, 7(78):4708, 2022. doi:10.21105/joss.04708.

18 Jean-Charles Régin. Generalized arc consistency for global cardinality constraint. In William J. Clancey and Daniel S. Weld, editors, *Proceedings of the Thirteenth National Conference on Artificial Intelligence and Eighth Innovative Applications of Artificial Intelligence Conference, AAAI 96, IAAI 96, Portland, Oregon, USA, August 4-8, 1996, Volume 1*, pages 209–215. AAAI Press / The MIT Press, 1996. URL: http://www.aaai.org/Library/AAAI/1996/aaai96-031.php.

19 Andrea Rendl and Matthias Prandtstetter. Constraint models for the container pre-marshaling problem. In *ModRef 2013: The Twelfth International Workshop on Constraint Modelling and Reformulation*, 2013.

20 Robert Tarjan. Sorting using networks of queues and stacks. *J. Assoc. Comput. Mach.*, 19:341–346, 1972. doi:10.1145/321694.321704.

21 Peter van Beek and Xinguang Chen. Cplan: A constraint programming approach to planning. In Jim Hendler and Devika Subramanian, editors, *Proceedings of the Sixteenth National Conference on Artificial Intelligence and Eleventh Conference on Innovative Applications of Artificial Intelligence, July 18-22, 1999, Orlando, Florida, USA*, pages 585–590. AAAI Press / The MIT Press, 1999. URL: http://www.aaai.org/Library/AAAI/1999/aaai99-083.php.

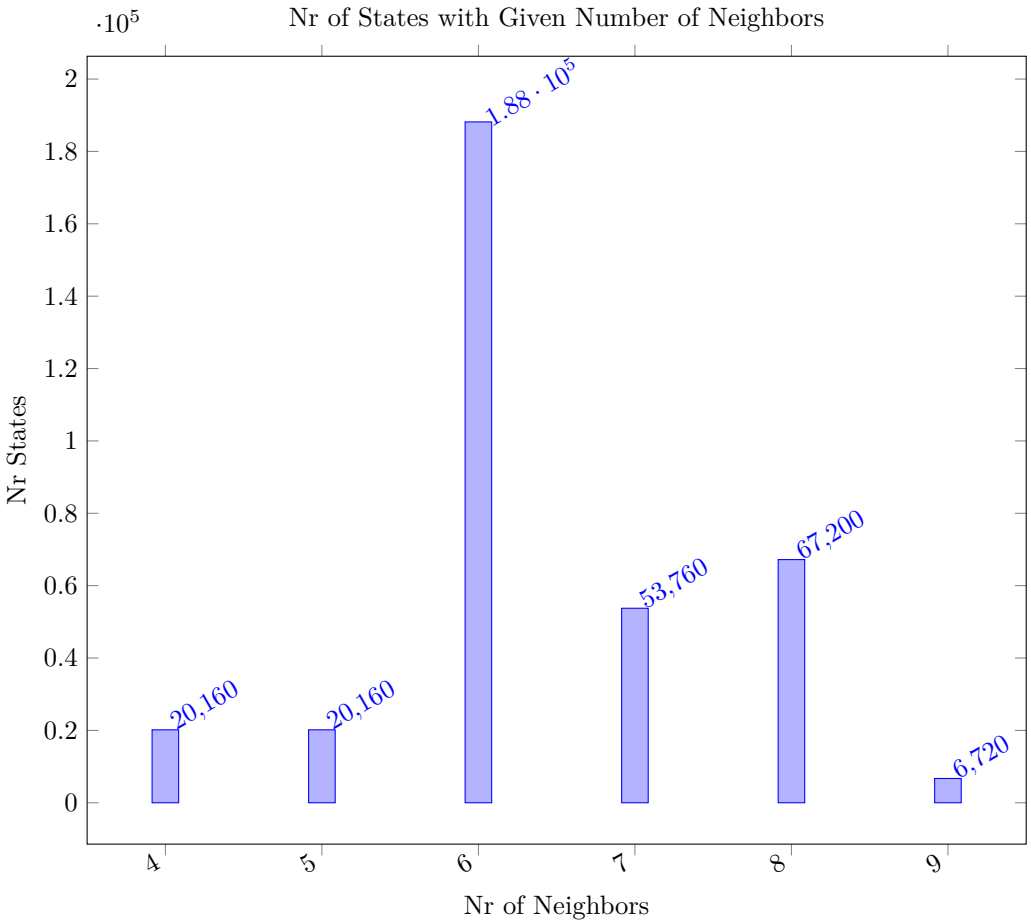
22 Adrian Wymann. The model railways shunting puzzles website. URL: https://www.wymann.info/ShuntingPuzzles/sw-inglenook.html.

23 Roland H. C. Yap, Wei Xia, and Ruiwei Wang. Generalized arc consistency algorithms for table constraints: A summary of algorithmic ideas. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 13590–13597. AAAI Press, 2020. URL: https://doi.org/10.1609/aaai.v34i09.7086, doi:10.1609/AAAI.V34I09.7086.

A Graph Analysis

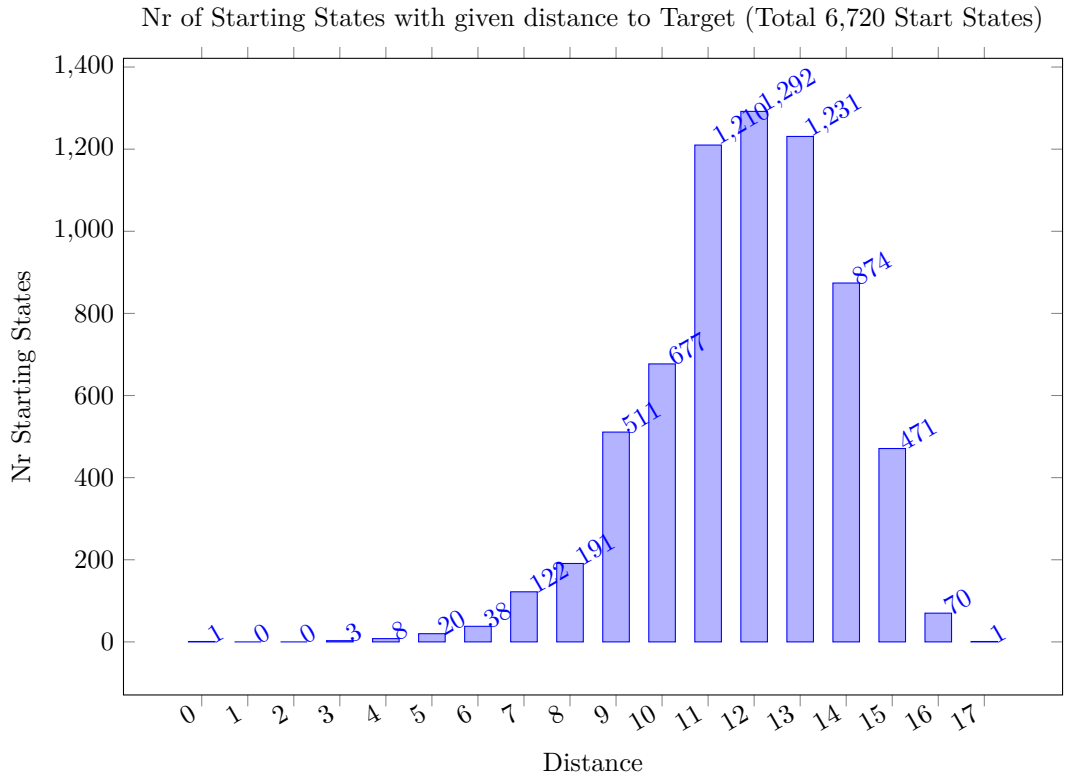
Figure 7 shows the distribution of states with a given number of neighbors for the default, eight wagon problem. In some states, we can only perform four legal moves, while the largest number of neighboring states is nine, but the most common number is six. This means that in each state we have between four and nine choices, so that the number of paths of a given length grows exponentially with the length.

Figure 7 Distribution of States with Given Number of Neighbors for Eight Wagon Problem



We can also investigate how many starting states have a given minimal distance to a target state in the graph. Figure 8 shows the distribution for the default problem with eight wagons. There is one starting state "12345/xxx" which is also a valid target state, and therefore has distance zero, otherwise we need at least three and at most 17 moves to reach a target state, with the most common distances being 11, 12, and 13. We use this information in the experiments section to generate problem instances with the minimal required number of moves ranging from 3 to 17.

Figure 8 Distribution of Instances by Minimal Number of Moves, Default Eight Wagon Problem



B More Results for State-Based Model

Tables 10, 11, and 12 shows the results of the state-based model for five, seven, and nine wagon problems.

Table 10 State Model - Time (in Seconds) to Find First Solution for 5 Wagons

			MiniZinc				Java	
Config	Nr Moves	Nr Sols	Chuffed	CP-Sat	Gecode	CPO	Choco	Jacop
12435	5	4	0.45	0.40	1.34	0.28	0.27	0.18
14235	7	2	0.49	0.40	1.13	0.13	0.17	0.26
31245	8	2	0.53	0.43	1.17	0.14	0.21	0.41
21345	9	20	0.54	0.46	1.22	0.15	0.17	0.47
32145	10	20	0.61	0.51	1.27	0.16	0.17	0.48
21354	11	88	0.63	0.67	1.33	0.16	0.18	0.49

Table 11 State Model - Time (in Seconds) to Find First Solution for 7 Wagons

			MiniZinc				Java	
Config	Nr Moves	Nr Sols	Chuffed	CP-Sat	Gecode	CPO	Choco	Jacop
5x 1234x	3	1	6.84	7.45	88.19	5.24	-	1.56
x4 123x5	4	1	7.60	7.58	104.49	6.08	-	2.06
xx 12435	5	4	9.14	9.31	120.35	6.20	-	3.21
5x 12x43	6	1	10.02	9.57	150.29	6.78	-	7.08
xx 14235	7	1	10.70	9.69	169.69	7.37	-	20.62
xx 31245	8	1	12.43	11.08	217.98	7.86	-	88.74
xx 21345	9	10	13.05	11.53	230.79	8.39	-	271.38
xx 21435	10	4	13.86	11.72	247.02	8.99	-	458.47
xx 32145	11	22	15.49	13.83	281.56	9.61	-	570.27
5x x4321	12	9	16.22	13.98	202.99	10.24	-	772.13
2x 543x1	13	34	16.83	14.57	323.98	10.91	-	1,007.58
1x x5432	14	41	17.94	16.12	341.56	11.57	-	1,131.86

Table 13 compares the time needed by the best solvers to prove infeasibility of instances where the given path length is too small by one. This clearly is the worst case situation. But all three solvers, (Chuffed, CP-Sat and CPO) are able to detect infeasibility just by propagation, with execution times very similar to the time needed to find a solution for the feasible problem with increased path length. This result is important if we want to prove optimality, but it requires the precomputation of the complete state graph.

The results shown are for satisfiability only, where for each instance we ask for a solution with the required minimal path length. For a new problem size, we will not know the optimal path length a priori, and have to solve an optimization problem. Table 14 shows some basic results. We compare two solvers, CPO with a bottom-up approach, and Chuffed with a top-down approach. In the bottom-up model, we start with the shortest possible path length, and try the model for that length. If it succeeds, then we have an optimal solution. If it fails, we increase the path length, until we find a solution. For the top-down approach we start with an a priori upper bound, and on finding a solution shorten the path by assigning more

Table 12 State Model - Time (in seconds) to Find First Solution for 9 Wagons (Parameters 9/5/5/4/4/4/3)

Config	Nr Moves	Nr Sols	Chuffed	CP-Sat	Gecode	CPO	Choco	Jacop
x5xx 1234x	3	1	106.57	98.36	-	101.77	-	24.95
xx4x 1235x	4	1	111.46	99.99	-	113.44	-	36.76
xxxx 12435	5	2	143.41	123.13	-	119.58	-	65.98
xxxx 13245	6	6	143.88	125.83	-	133.91	-	245.53
xxxx 41235	7	1	158.86	128.58	-	149.53	-	1,212.94
xxxx 31245	8	3	205.95	152.04	-	171.10	-	4,153.93
xxxx 21345	9	10	211.06	153.71	-	165.42	-	6,776.66
xxxx 43215	10	24	202.11	156.68	-	180.67	-	TO
xxxx 21543	11	34	222.67	181.24	-	194.75	-	TO
5xxx x3241	12	14	238.74	185.82	-	215.07	-	TO
5xxx 432x1	13	87	263.80	190.19	-	233.21	-	TO
1xxx 543x2	14	67	255.92	197.40	-	270.82	-	TO

Table 13 State Model - Time (in Seconds) to Show Infeasibility of Relaxed Problem (NrMoves-1 Steps) for 8 Wagons

Config	Nr Moves	Chuffed	CP-Sat	CPO
x5x 1234x	3	17.32	14.19	9.68
5xx 1234x	4	20.73	18.28	11.02
xxx 12435	5	21.34	18.81	14.46
3xx 12x45	6	25.93	22.85	14.23
xxx 14235	7	30.02	23.69	16.26
5xx 1243x	8	30.71	23.67	17.63
xxx 21345	9	35.39	27.87	20.29
xxx 41235	10	36.24	28.89	20.06
xxx 32145	11	40.71	29.22	21.88
xxx 34125	12	44.28	34.10	23.16
5xx x4231	13	45.47	36.66	24.72
5xx x3241	14	47.04	36.84	26.69
2xx 4351x	15	52.82	39.48	28.95
1xx x3254	16	57.49	44.51	30.52
21x xx543	17	59.60	42.59	33.69

end values. This works very well for Chuffed, as its default search typically finds solution with the largest number of end-values at the end of the path.

Table 14 State Model - Time (in Seconds) to Find Optimal Solution for 8 Wagons

Config	Optimal Nr Moves	MiniZinc Chuffed Top-Down	Java CPO Bottom-up
x5x 1234x	3	65.77	15.04
5xx 1234x	4	63.99	28.95
xxx 12435	5	63.96	43.10
3xx 12x45	6	57.44	61.03
xxx 14235	7	62.20	77.77
5xx 1243x	8	64.58	103.73
xxx 21345	9	65.25	135.91
xxx 41235	10	63.93	164.26
xxx 32145	11	61.14	208.66
xxx 34125	12	62.49	233.69
5xx x4231	13	67.67	237.99
5xx x3241	14	60.91	277.59
2xx 4351x	15	58.32	331.38
1xx x3254	16	58.99	371.84
21x xx543	17	59.77	412.55

Note that both approaches can be significantly improved by exploiting the fact that we achieve domain consistency. For the top-down model, we can generate a custom search routine which forces the tail of the path variables to the end value. For the bottom-up model, we can try a binary search for the optimal value, or run models with multiple path-length limits in parallel.

C More Results for the String-Based Model

Tables 15 and 16 show the results for the string-based model for 5 and 7 wagons. We did not run problem instances for nine or ten wagons.

Table 15 String Based-Model - Time (in seconds) and Number of Choices with Z3 for 5 Wagons

Config	Nr Moves	Time	Nr Decisions
12435	5	0.25	318
14235	7	0.77	6,684
31245	8	2.10	17,045
21345	9	5.02	25,509
32145	10	33.18	53,804
21354	11	325.00	133,841

Table 16 String Based-Model - Time (in seconds) and number of Choices with Z3 for 7 Wagons

Config	Nr Moves	Time	Nr Decisions
5x 1234x	3	0.25	1,347
x4 123x5	4	0.26	1,197
xx 12435	5	0.26	1,757
5x 12x43	6	16.11	55,683
xx 14235	7	0.98	8,947
xx 31245	8	4.12	28,800
xx 21345	9	3.63	29,348
xx 21435	10	24.71	57,695
xx 32145	11	273.38	192,548
5x x4321	12	TO	-
2x 543x1	13	TO	-
1x x5432	14	TO	-

D More Results for Position-Based Model

Tables 17, 18, 19, and 20 show the results of the position-based model on the smaller problem sizes five, six, and seven, as wells as for nine wagons. Even for the smallest sizes, Gecode is not able to solve all instances.

Table 17 Position Model - Time (in seconds) to Find First Solution for 5 Wagons, All Solvers in MiniZinc

Config	Nr Moves	Chuffed		CP-Sat		Cplex		Gecode	
		Time	Nodes	Time	Failures	Time	Nodes	Time	Nodes
12435	5	0.78	1,154	1.04	26	0.82	0	1.19	92,072
14235	7	0.63	9,630	0.62	183	1.19	0	11.52	3,710,497
31245	8	1.00	19,323	0.68	656	1.79	718	TO	>88,826,386
21345	9	0.74	14,809	0.75	164	1.06	0	TO	>80,964,084
32145	10	1.65	34,928	0.89	2,091	2.23	1,298	TO	>76,396,792
21354	11	8.08	105,598	0.87	582	11.36	8,413	TO	>67,476,721

Table 18 Position Model - Time (in seconds) to Find First Solution for 6 Wagons, All Solvers in MiniZinc

Config	Nr Moves	Chuffed		CP-Sat		Cplex		Gecode	
		Time	Nodes	Time	Failures	Time	Nodes	Time	Nodes
5 1234x	3	0.37	101	0.46	0	0.46	0	0.34	233
5 123x4	4	0.36	810	0.41	0	0.52	0	3.83	1,205,710
x 12435	5	0.37	1,546	0.52	123	0.74	0	6.11	2,036,009
5 124x3	6	0.43	4,127	0.56	272	0.93	0	295.10	104,278,426
x 14235	7	1.04	19,807	0.64	455	1.64	153	TO	>102,087,090
x 31245	8	4.92	71,795	0.71	763	10.98	9,621	TO	>90,663,333
x 21345	9	1.75	40,181	0.79	943	2.34	180	TO	>85,931,315
x 32145	10	24.58	207,624	0.89	630	129.05	47,912	TO	>80,075,211
x 21354	11	8.03	95,531	1.02	4,118	231.35	69,219	TO	>64,320,990
1 x5432	12	7.18	110,257	1.04	3,678	287.69	85,043	TO	>71,438,350

Table 19 Position Model - Time (in seconds) to Find First Solution for 7 Wagons, All Solvers in MiniZinc

Config	Nr Moves	Chuffed		CP-Sat		Cplex		Gecode	
		Time	Nodes	Time	Failures	Time	Nodes	Time	Nodes
5x 1234x	3	0.35	67	0.40	0	0.42	0	0.34	543
x4 123x5	4	0.33	410	0.44	0	0.61	0	5.40	2,079,016
xx 12435	5	0.39	2,127	0.52	112	0.80	0	50.12	19,008,873
5x 12x43	6	0.69	13,556	0.61	296	1.42	246	TO	>114,384,439
xx 14235	7	0.99	21,672	0.67	267	3.75	3,368	TO	>112,096,981
xx 31245	8	1.09	23,758	0.76	651	29.59	19,764	TO	>102,854,194
xx 21345	9	1.86	46,093	0.87	3,512	113.01	35,339	TO	>90,796,428
xx 21435	10	2.00	55,888	1.03	3,408	81.04	29,319	TO	>86,480,756
xx 32145	11	4.55	95,084	1.14	3,501	38.88	16,438	TO	>65,743,243
5x x4321	12	4.74	104,841	1.25	5,789	TO	>70,382	TO	>28,903,445
2x 543x1	13	13.24	186,311	1.41	7,944	TO	>69,519	TO	>58,181,117
1x x5432	14	60.76	488,725	2.40	2,958	TO	>55,884	TO	>52,327,002

Table 20 Position Model - Time (in seconds) to Find First Solution for 9 Wagons, All Solvers in MiniZinc

Config	Nr Moves	Chuffed		CP-Sat		Cplex		Gecode	
		Time	Nodes	Time	Failures	Time	Nodes	Time	Nodes
x5xx 1234x	3	3.24	249	1.10	0	0.72	0	1.23	1,541
xx4x 1235x	4	0.38	1,113	0.45	0	0.81	0	204.96	85,464,241
xxxx 12435	5	0.47	5,073	0.54	140	0.82	0	84.15	29,956,721
xxxx 13245	6	0.50	6,756	0.66	569	1.28	0	TO	>97,475,197
xxxx 41235	7	0.72	14,671	0.76	227	9.15	6,201	TO	>86,394,700
xxxx 31245	8	1.25	28,077	0.85	1,377	171.77	42,499	TO	>76,724,666
xxxx 21345	9	1.65	39,360	0.90	826	280.62	72,615	TO	>67,941,473
xxxx 43215	10	1.63	42,822	1.06	2,567	TO	>74,139	TO	>58,438,772
xxxx 21543	11	3.21	78,377	1.32	8,122	108.31	19,513	TO	>50,167,323
5xxx x3241	12	6.79	125,633	2.64	15,140	TO	>65,585	TO	>46,353,858
5xxx 432x1	13	59.36	578,932	3.13	21,527	TO	>58,024	TO	>50,691,919
1xxx 543x2	14	159.02	1,177,650	3.19	21,290	TO	>51,051	TO	>42,295,170

D.1 Impact of Number of Worker Threads on CP-Sat Results for the Position-Based Model

Tables 21 and 22 compare the results for CP-Sat on instances of sizes 8 and 10 with varying numbers of worker threads. While the results indicate that more instances are solved fastest with eight worker threads, the results for two to eight workers differ only slightly. On the other hand, results with only one worker thread for the larger instances are much weaker. This is a common outcome for CP-Sat for other problems, CP-Sat seems optimized to use at least two workers in parallel.

Table 21 Position Model - Time (in seconds) to Find First Solution for 8 Wagons, CP-Sat in MiniZinc, 1-8 Worker Threads, Fastest Run in Bold

		Number of Worker Threads							
Config	Nr Moves	1	2	3	4	5	6	7	8
x5x 1234x	3	0.50	0.41	0.39	0.38	0.38	0.50	0.38	**0.36**
5xx 1234x	4	**0.39**	0.45	0.42	0.42	0.40	**0.39**	0.53	0.40
xxx 12435	5	0.66	0.51	0.50	0.51	0.52	**0.48**	0.59	**0.48**
3xx 12x45	6	0.63	0.61	0.59	0.56	0.56	0.56	0.65	**0.55**
xxx 14235	7	0.79	0.77	0.68	0.65	0.66	0.64	0.66	**0.61**
5xx 1243x	8	1.26	0.73	0.73	0.78	0.74	0.71	0.70	**0.66**
xxx 21345	9	1.21	0.82	0.80	0.81	0.80	**0.76**	0.79	0.78
xxx 41235	10	1.05	0.89	0.85	0.92	0.88	0.84	0.83	**0.81**
xxx 32145	11	4.02	1.08	0.98	1.07	1.07	1.02	1.04	**0.94**
xxx 34125	12	6.65	**1.20**	**1.20**	1.28	**1.20**	1.37	1.24	**1.20**
5xx x4231	13	8.65	2.43	2.34	2.30	2.41	**1.36**	1.48	1.42
5xx x3241	14	20.34	2.74	1.74	2.52	**1.60**	3.06	2.40	2.52
2xx 4351x	15	27.26	2.81	3.10	2.69	2.66	2.81	2.63	**2.54**
1xx x3254	16	108.55	3.22	3.49	4.91	3.49	3.21	**2.87**	2.93
21x xx543	17	174.02	4.31	3.74	**3.31**	4.17	3.55	3.40	4.15

Table 22 Position Model - Time (in seconds) to Find First Solution for 10 Wagons, CP-Sat in MiniZinc, 1-8 Worker Threads, Fastest Run in Bold

		Number of Worker Threads							
Config	Nr Moves	1	2	3	4	5	6	7	8
xx5xx 1234x	3	0.41	0.43	0.42	0.42	0.42	0.42	**0.39**	**0.39**
xxx5x 123x4	4	0.46	0.49	0.46	0.48	0.46	0.57	0.46	**0.44**
xxxxx 12435	5	0.64	0.57	0.58	0.56	0.58	0.57	0.55	**0.53**
xxxxx 13245	6	0.73	0.68	0.67	0.66	0.65	0.63	0.62	**0.60**
xxxxx 41235	7	0.97	0.76	0.76	0.73	0.76	0.78	0.72	**0.70**
xxxxx 31245	8	1.20	0.91	0.86	0.86	0.86	0.85	0.80	**0.79**
xxxxx 21345	9	1.87	1.02	0.96	1.00	0.99	1.01	0.93	**0.88**
xxxxx 43215	10	2.74	1.13	1.07	1.03	1.14	1.01	**0.96**	0.99
xxxxx 21543	11	4.90	1.27	1.24	1.26	1.21	1.19	1.40	**1.12**
5xxxx x2341	12	14.12	1.62	**1.40**	1.78	1.68	1.79	1.54	1.62
5xxxx 32x41	13	7.94	**1.43**	3.64	2.94	3.00	2.76	1.77	2.74
2xxxx 5314x	14	377.12	3.06	2.17	3.26	3.12	3.23	3.11	**1.60**
1xxxx 543x2	15	1245.39	3.53	3.32	3.38	1.96	3.89	3.09	**1.82**
2154x xxx3x	16	TO	3.69	4.68	1.92	3.75	**1.63**	5.83	3.59

E Split State Model

We now briefly present an additional model, which is still state based, but which uses a different state representation, using two variables to describe each state.

The state-based model in Section 3.1 used a unique identifier for each state, and expressed the possible moves as a table with two columns. For bigger problem sizes, the number of states is quite large (see Table 1), and we may want to explore a different state representation. We have described in Section 2 how the states are created as the Cartesian product of distributions and permutations. We can choose a different state representation with two variables, one which describes the distribution used, and one which describes the permutation used in the state. In our constraint model we now use two variables, $xD[i]$ for the distribution, and $xP[i]$ for the permutation used in step i. The moves are represented as a table with four columns, the total number of rows in the table does not change compared to the state-based model, as it is the number of feasible moves between any two states.

Program 9 shows the resulting MiniZinc program, with the moves loaded as an array with four columns. We again introduce an artificial target state which can only be reached from the states satisfying the target condition. Two values each are used to identify the start and target state, and are given as input data. The requested path length *size* is also an input parameter.

Figure 9 MiniZinc Program for Split State-Based Model

```
include "globals.mzn";
int:startD;
int:startP;
int:targetD;
int:targetP;
int:size;
array[int,1..4] of int:moves;


array[1..size] of var 0..targetD:xD;
array[1..size] of var 0..targetP:xP;

constraint xD[1] = startD;
constraint xP[1] = startP;
constraint xD[size] = targetD;
constraint xP[size] = targetP;
constraint forall(i in 1..size-1)(table([xD[i],xP[i],xD[i+1],xP[i+1]],moves));

solve satisfy;
```

Tables 23 to 26 show the results of running this model on problem instances with five to eight wagons, using the MiniZinc model given. It seems that Gecode prefers this model over the state-based model, while Chuffed and CP-Sat perform much worse, and are not able to find solutions within the timeout limit for instances that they can solve easily with the state-based model. As this split-state model also achieves generalized arc-consistency through propagation alone, the difference is not due to excessive search, but only caused by a different sequence of steps used to propagate the `table` constraints.

Table 23 Split Model - Time to Find First Solution for 5 Wagons

			MiniZinc				Java	
Config	Nr Moves	Nr Sols	Chuffed	CP-Sat	Gecode	CPO	Choco	Jacop
12435	5		0.89	0.67	1.53	0.31		
14235	7		0.73	0.73	1.33	0.17		
31245	8		0.77	0.84	1.34	0.17		
21345	9		0.80	1.36	1.35	0.19		
32145	10		1.20	5.33	1.39	0.22		
21354	11		1.06	15.71	1.51	0.24		

Table 24 Split Model - Time to Find First Solution for 6 Wagons

			MiniZinc				Java	
Config	Nr Moves	Nr Sols	Chuffed	CP-Sat	Gecode	CPO	Choco	Jacop
5 1234x	3		3.08	3.29	4.46	1.45		
5 123x4	4		3.80	4.34	4.90	1.48		
x 12435	5		4.08	4.40	5.20	1.56		
5 124x3	6		4.42	4.80	5.50	1.69		
x 14235	7		5.23	6.33	6.71	1.82		
x 31245	8		5.91	8.08	6.89	2.04		
x 21345	9		6.51	23.67	7.35	2.22		
x 32145	10		15.06	58.24	8.46	2.48		
x 21354	11		9.13	74.04	10.41	4.63		
1 x5432	12		42.43	98.40	9.24	3.73		

Table 25 Split Model - Time (in seconds) to Find First Solution for 7 Wagons

			MiniZinc				Java	
Config	Nr Moves	Nr Sols	Chuffed	CP-Sat	Gecode	CPO	Choco	Jacop
5x 1234x	3		11.94	13.44	16.18	5.61		
x4 123x5	4		13.14	13.96	16.79	6.00		
xx 12435	5		15.72	17.31	19.54	7.21		
5x 12x43	6		16.84	18.44	20.79	7.02		
xx 14235	7		17.77	20.62	22.22	7.49		
xx 31245	8		20.58	27.72	25.78	7.96		
xx 21345	9		21.67	65.34	27.33	8.79		
xx 21435	10		24.12	164.34	33.59	9.83		
xx 32145	11		103.96	244.91	52.49	26.30		
5x x4321	12		722.45	757.53	60.73	173.40		
2x 543x1	13		3,365.28	544.00	41.99	120.39		
1x x5432	14		TO	TO	625.49	251.43		

Table 26 Split Model - Time (in seconds) to Find First Solution for 8 Wagons

			MiniZinc				Java	
Config	Nr Moves	Nr Sols	Chuffed	CP-Sat	Gecode	CPO	Choco	Jacop
x5x 1234x	3		29.63	33.25	39.68	15.52		
5xx 1234x	4		39.62	34.91	45.02	16.60		
xxx 12435	5		48.15	43.81	53.44	19.51		
3xx 12x45	6		46.43	46.37	58.73	19.09		
xxx 14235	7		66.50	50.17	63.68	21.34		
5xx 1243x	8		66.14	77.75	70.88	22.20		
xxx 21345	9		259.21	123.75	78.65	25.58		
xxx 41235	10		186.72	301.84	89.49	27.16		
xxx 32145	11		TO	734.58	112.09	51.85		
xxx 34125	12		TO	1,579.99	306.06	377.39		
5xx x4231	13			TO	969.98	393.78		
5xx x3241	14					2,245.55		
2xx 4351x	15					1,385.17		
1xx x3254	16					TO		
21x xx543	17					TO		

F Position Model Variations

In this section we discuss potential variations of the position based model. In the version defined in Section 3.3, not all constraints are strictly required, the `global_cardinality` constraint and the implications are redundant constraint intended to help with propagation. We should check that these constraints really help with solving the problem, and that other potential redundant constraints are not helping. We focus on the standard problem size of 8 wagons, for the simpler instances with 5, 6, or seven wagons the impact of the redundant constraints is less noticeable. A summary of the results for CP-Sat only were given in Table 9, where we compare different variants to our default model. The results indicate that the global cardinality constraint is very important to the model, while the different implication have a much smaller impact.

Results in this section should be compared to Table 7.

F.1 Adding More Implications

The position model above uses some implications to state that any zero values are at the front of shunts A,B, and C, and at the back of shunt X. If a zero value occurs in some position, then all values to the left (resp. right for X) must also be zero. There is a corresponding rule stating that if a non-zero value occurs in a shunt, then all values to the right (resp. left for shunt X) must also be non-zero. As a constraint this is weaker, as it uses disequality constraints for the check, and for the implied condition.

```
constraint forall(i in 1..size,j in 1..lengthA-1)(A[i,j] != 0 -> A[i,j+1] != 0);
constraint forall(i in 1..size,j in 1..lengthB-1)(B[i,j] != 0 -> B[i,j+1] != 0);
constraint forall(i in 1..size,j in 1..lengthC-1)(C[i,j] != 0 -> C[i,j+1] != 0);
constraint forall(i in 1..size,j in 2..lengthX)(X[i,j] != 0 -> X[i,j-1] != 0);
```

Table 27 shows the results for the standard problem size. For some instances, results are improved, but for others, it takes more time and nodes to find the solution. This can be explained by changes in the search tree exploration in the updated program.

F.2 Pure Model without Global Cardinality

The `global_cardinality` constraint in the position model is used to state that we use the correct number of the values 0..6 describing each state. But for the first state this is given by the input data, and the predicate move has an invariant stating that each move does not change the number of of values used from one state to the next. We can therefore remove the `global_cardinality` constraint without affecting the solution space. This change will affect the propagation, we expect less propagation without it, but it will also affect the search tree traversal, as variables will be selected in a different order. We want to check whether this changes the results.

```
int:size;
int:nrWagons;
int:lengthA=5;
int:lengthB=3;
int:lengthC=3;
int:lengthX=3;
int:lengthTotal = lengthA+lengthB+lengthC+lengthX;
int:nrZero = lengthTotal-nrWagons;
int:nrDontCare = nrWagons-5;

set of int:Run = 1..size;
set of int:Dom = 0..6;
array[int] of int:startA;
array[int] of int:startB;
array[int] of int:target=[1,2,3,4,5];

```

Table 27 Stronger Position Model - Time (in seconds) to Find First Solution for 8 Wagons, All Solvers in MiniZinc

Config	Nr Moves	Chuffed		CP-Sat		Cplex		Gecode	
		Time	Nodes	Time	Failures	Time	Nodes	Time	Nodes
x5x 1234x	3	0.30	121	0.34	0	0.35	0	1.20	1,275
5xx 1234x	4	0.33	811	0.39	0	0.51	0	28.95	15,460,952
xxx 12435	5	0.35	1,249	0.45	196	0.70	0	60.03	28,095,542
3xx 12x45	6	0.42	4,749	0.53	261	0.77	0	TO	
xxx 14235	7	0.50	9,067	0.60	592	1.79	708	TO	
5xx 1243x	8	2.54	45,343	0.67	677	22.64	14,791	TO	
xxx 21345	9	0.71	17,763	0.71	1,500	23.93	10,616	TO	
xxx 41235	10	1.99	58,877	0.98	1,059	281.14	104,335	TO	
xxx 32145	11	2.96	77,743	1.07	3,498	TO		TO	
xxx 34125	12	4.31	111,796	1.09	6,414	TO		TO	
5xx x4231	13	16.49	226,184	1.14	5,428	TO		TO	
5xx x3241	14	56.22	502,488	2.32	17,235	TO		TO	
2xx 4351x	15	22.97	338,606	2.49	20,015	TO		TO	
1xx x3254	16	9.14	205,275	3.66	36,823	TO		TO	
21x xx543	17	116.51	934,771	2.92	29,209	TO		TO	

```
17 array[Run,1..lengthA] of var Dom:A;
18 array[Run,1..lengthB] of var Dom:B;
19 array[Run,1..lengthC] of var Dom:C;
20 array[Run,1..lengthX] of var Dom:X;
21 array[Run,1..lengthTotal] of var Dom:S;
22
23 constraint forall(i in 1..size)(row(S,i) = row(A,i)++row(B,i)++row(C,i)++row(X,i));
24 constraint row(A,1) = startA;
25 constraint row(B,1) = startB;
26 constraint row(C,1) = [0,0,0];
27 constraint row(X,1) = [0,0,0];
28 constraint row(A,size) = target;
29 constraint forall(i in 1..size,j in 2..lengthA)(A[i,j] = 0 -> A[i,j-1] = 0);
30 constraint forall(i in 1..size,j in 2..lengthB)(B[i,j] = 0 -> B[i,j-1] = 0);
31 constraint forall(i in 1..size,j in 2..lengthC)(C[i,j] = 0 -> C[i,j-1] = 0);
32 constraint forall(i in 1..size,j in 1..lengthX-1)(X[i,j] = 0 -> X[i,j+1] = 0);
33 constraint forall(i in 1..size-1)
34     (move(row(A,i),row(B,i),row(C,i),row(X,i),row(A,i+1),row(B,i+1),row(C,i+1),row(X,i+1)));
35
36 solve satisfy;
```

Results are shown in Table 28.

Table 28 Position Pure Model - Time (in seconds) to Find First Solution for 8 Wagons, All Solvers in MiniZinc

Config	Nr Moves	Chuffed		CP-Sat		Cplex		Gecode	
		Time	Nodes	Time	Failures	Time	Nodes	Time	Nodes
x5x 1234x	3	2.92	183	1.01	0	0.70	0	1.65	36,657
5xx 1234x	4	0.30	551	0.37	12	0.52	0	1.45	348,443
xxx 12435	5	0.35	3,215	0.41	59	0.59	0	0.99	36,978
3xx 12x45	6	0.32	1,497	0.43	401	1.58	2,582	1.97	473,502
xxx 14235	7	0.42	8,385	0.56	1,437	16.96	16,965	8.52	3,860,578
5xx 1243x	8	0.83	21,373	0.85	2,548	52.94	46,326	35.00	16,433,324
xxx 21345	9	0.74	26,551	1.60	10,192	51.40	39,057	61.43	27,526,468
xxx 41235	10	0.82	31,278	1.84	9,441	230.36	145,359	TO	
xxx 32145	11	1.32	51,921	6.33	34,759	TO		TO	
xxx 34125	12	6.16	125,983	7.55	52,448	TO		TO	
5xx x4231	13	20.84	215,846	13.96	93,353	TO		TO	
5xx x3241	14	20.57	279,699	66.44	288,896	TO		TO	
2xx 4351x	15	13.47	254,686	49.23	228,767	TO		TO	
1xx x3254	16	31.77	400,527	186.11	667,133	TO		TO	
21x xx543	17	58.59	587,110	TO		TO		TO	

F.3 Even Purer Model

Once we remove the `global_cardinality` constraint, we do not really need the concatenated array S of all variables per state. This cleans up the code a bit more.

```
int:size;
int:nrWagons;
int:lengthA=5;
int:lengthB=3;
int:lengthC=3;
int:lengthX=3;
int:lengthTotal = lengthA+lengthB+lengthC+lengthX;
int:nrZero = lengthTotal-nrWagons;
int:nrDontCare = nrWagons-5;

set of int:Run = 1..size;
set of int:Dom = 0..6;
array[int] of int:startA;
array[int] of int:startB;
array[int] of int:target=[1,2,3,4,5];

array[Run,1..lengthA] of var Dom:A;
array[Run,1..lengthB] of var Dom:B;
array[Run,1..lengthC] of var Dom:C;
array[Run,1..lengthX] of var Dom:X;

constraint row(A,1) = startA;
constraint row(B,1) = startB;
constraint row(C,1) = [0,0,0];
constraint row(X,1) = [0,0,0];
constraint row(A,size) = target;
constraint forall(i in 1..size,j in 2..lengthA)(A[i,j] = 0 -> A[i,j-1] = 0);
constraint forall(i in 1..size,j in 2..lengthB)(B[i,j] = 0 -> B[i,j-1] = 0);
constraint forall(i in 1..size,j in 2..lengthC)(C[i,j] = 0 -> C[i,j-1] = 0);
constraint forall(i in 1..size,j in 1..lengthX-1)(X[i,j] = 0 -> X[i,j+1] = 0);
constraint forall(i in 1..size-1)
    (move(row(A,i),row(B,i),row(C,i),row(X,i),row(A,i+1),row(B,i+1),row(C,i+1),row(X,i+1)));

solve satisfy;
```

The results for 8 wagons are shown in Table 29.

Table 29 Position Purer Model - Time (in seconds) to Find First Solution for 8 Wagons, All Solvers in MiniZinc

Config	Nr Moves	Chuffed		CP-Sat		Cplex		Gecode	
		Time	Nodes	Time	Failures	Time	Nodes	Time	Nodes
x5x 1234x	3	2.88	187	0.98	0	0.66	0	1.63	9,356
5xx 1234x	4	0.28	771	0.36	12	0.55	0	1.16	97,302
xxx 12435	5	0.32	1,398	0.42	303	0.64	0	1.27	219,092
3xx 12x45	6	0.35	5,440	0.46	315	1.76	2,582	2.10	581,845
xxx 14235	7	0.36	3,309	0.61	1,638	19.15	16,965	8.34	3,971,408
5xx 1243x	8	0.55	15,246	0.55	1,448	55.87	46,326	52.27	25,535,070
xxx 21345	9	0.67	25,591	1.19	7,318	52.14	39,057	233.24	108,415,395
xxx 41235	10	0.63	23,008	2.20	15,038	225.56	145,359	TO	
xxx 32145	11	1.63	51,901	5.03	34,649	TO		TO	
xxx 34125	12	5.65	136,088	13.65	66,034	TO		TO	
5xx x4231	13	17.81	268,835	10.21	71,255	TO		TO	
5xx x3241	14	142.44	1,250,421	50.15	296,312	TO		TO	
2xx 4351x	15	81.53	814,305	30.68	167,454	TO		TO	
1xx x3254	16	TO		199.31	726,025	TO		TO	
21x xx543	17	73.94	849,959	262.22	862,559	TO		TO	

F.4 Purest Position Model

The final variant we consider also removes the implications from the model, we are left with only the move constraints between the variables of two consecutive states. The results are shown in Table 30. Results for CP-Sat are disappointing, it no longer finds solutions for all instances within the timeout.

Table 30 Position Purest Model - Time (in seconds) to Find First Solution for 8 Wagons, All Solvers in MiniZinc

Config	Nr Moves	Chuffed		CP-Sat		Cplex		Gecode	
		Time	Nodes	Time	Failures	Time	Nodes	Time	Nodes
x5x 1234x	3	0.27	147	0.32	0	0.73	0	1.65	18,593
5xx 1234x	4	0.28	681	0.37	56	0.51	0	0.35	19,708
xxx 12435	5	0.31	1,648	0.37	153	0.69	0	1.48	276,717
3xx 12x45	6	0.40	8,351	0.45	444	1.52	1,276	1.56	239,387
xxx 14235	7	0.69	18,968	0.56	1,548	3.71	5,952	16.90	7,822,881
5xx 1243x	8	0.72	22,126	0.80	3,204	11.73	11,376	21.21	9,529,100
xxx 21345	9	0.79	26,799	1.97	13,594	41.57	36,377	TO	
xxx 41235	10	1.92	58,827	2.99	21,745	28.41	25,257	TO	
xxx 32145	11	5.96	148,427	7.25	53,705	TO		TO	
xxx 34125	12	5.00	122,742	13.11	83,477	TO		TO	
5xx x4231	13	10.89	227,354	13.17	79,778	TO		TO	
5xx x3241	14	44.64	623,231	61.70	264,293	TO		TO	
2xx 4351x	15	14.44	291,429	171.75	363,160	TO		TO	
1xx x3254	16	151.01	1,528,946	TO		TO		TO	
21x xx543	17	215.57	2,117,069	54.82	256,328	TO		TO	