Answer Set Programming Model for Automated Fulfillment Warehouse

Written by Youmi Koh
Arizona State University
ykoh7@asu.edu

Abstract

This report presents the Automated Fulfillment Warehouse Scenario and its representation as a planning problem using Answer Set Programming. It provides background on the methodology used in encoding the scenario, and discusses the results from solving the program using clingo. The report highlights critical constraints for producing valid and optimal plans, and highlights opportunities for future work.

Problem Statement

The Automated Fulfillment Warehouse Scenario illustrates a warehouse environment staffed with robots that are tasked with delivering products on stock shelves to appropriate picking stations where orders are fulfilled. Orders can be composed of multiple product line items, and robots have various operational limitations. The objective is to devise a plan that fulfills all warehouse orders in the least number of time steps.

Project Background

The Automated Warehouse Scenario is a planning problem that can be represented using answer set programming (ASP), a declarative programming paradigm that models problems as logic programs, and represents them as a set of rules and constraints. The variables of the program are instantiated with values to produce a grounded program, which serves as the program's knowledge base for deducing the universe of solutions. ASP solvers ground programs, then compute answer sets, which consists of atoms that satisfy all rules and constraints of the program.

Rules and constraints are expressed using fluents, actions, and axioms. Fluents depict the state of the world at a given time, and can be influenced by actions and axioms. Actions are the events that can occur in the world, and can affect the state of the world. Axioms are the rules that govern the behaviour of the world, and can be used to constrain the state of the world and the actions that can occur. The program's solution space is reduced to produce a set of answer sets that satisfy all rules and constraints of the program.

Programs for planning problems are encoded with a starting state and a future goal state. An objective function is defined to evaluate the quality of a plan, and the aim finding plans that transforms the starting state into the goal state while optimizing the objective function.

In this scenario, the program is initialized with a starting instance state for all components of the warehouse, and the goal state is defined as a warehouse with all orders fulfilled. The makespan function is expressed as the number of time steps required to fulfill all orders, and optimal plans minimize the makespan value. In this project, we will use clingo as the ASP solver, to find these optimal plans.

Program Representation and Methodology

Inside the warehouse, the scenario starts at time `T=0` and the floor is represented in a grid format. Every cell on the grid floor is expressed as `node(N,(X,Y))` where `(X,Y)` are the cell coordinates, and symbolize a location of the warehouse floor where physical objects can be positioned. Certain cells of the warehouse floor are designated as highways and order picking stations, expressed as `highway(H,(X,Y))` and `pickingStation(PS,(X,Y))` respectively.

All robots `R` and shelves `S` are located at a node, and their position at time `T` is declared as `robotAt(R,(X,Y),T)` and `shelfAt(S,(X,Y),T)` respectively. Products can be stocked in multiple shelves, and inventory levels are expressed as `inventoryOn(S,(P,Q),T)` where `Q` denotes the quantity of product `P` stocked on shelf `S` at time `T`.

Robots are mobile, and can move to adjacent nodes in 4 cardinal directions: up, down, left and right. The movement of robot `R` at time `T` is expressed as `move(R,(DX,DY),T)` where `(DX,DY)` is a directional unit vector. Robots can pick up and put down shelves, which enables them to carry shelves as they move and relocate the shelf to a new position. The action of a robot picking up a shelf is expressed as `pickup(R,S,T)`, and the action of putting down a shelf is declared as `putdown(R,S,T)`. The boolean value of `B` in `carrying(B,R,S,T)` is set to `t` for true, when robot `R` is carrying shelf `S`, and `f` for false otherwise.

The warehouse must fulfill all orders at its designated picking station. Orders are composed of multiple line items that state the ordered units of a product, expressed as `lineItem(O,(P,U),T)` where `U` denotes the remaining number of fulfillable units of product `P` at time `T` on order `O`. The action of a robot delivering `D` number of products for an order line item is declared as `deliver(R,(O,P,D),T)`.

```
robot(R) :-
    init(object(robot,R),value(at,pair(X,Y))).
shelf(S) :-
    init(object(shelf,S),value(at,pair(X,Y))).
product(P,S) :-
    init(object(product,P),value(on,pair(S,Q))).
order(O,PS) :-
    init(object(order,O),value(pickingStation,PS)).
order(O,P,U) :-
    init(object(order,O),value(line,pair(P,U))).

robotAt(R,(X,Y),0) :-
    init(object(robot,R),value(at,pair(X,Y))).
shelfAt(S,(X,Y),0):-
    init(object(shelf,S),value(at,pair(X,Y))).
inventoryOn(S,(P,Q),0) :-
    init(object(product,P),value(on,pair(S,Q))).
lineItem(O,(P,U),0) :-
    init(object(order,O),value(line,pair(P,U))).
```

Figure 1: Initial state

The primary challenges with this project was to represent the scenario as a program with a reasonable starting search space. In the following sections, there is heavy focus on grounding, generative axioms, and their effects on constraints and satisfiability of the program.

Instantiation and Ground Terms

The warehouse program is initialized with a given instance, which instantiates a set of ground terms that represent the starting state of the warehouse. As shown in Figure 1, this includes constant objects as well as fluents that represent the state of the world at time `T=0`. This set of ground terms are the basis for generating our program's search space.

There are several instance rules that propagate both a constant object and a fluent, such as `robot(R)` and `robotAt(R,(X,Y),0)`. In order to minimize complexity and simplify rules and constraints of the program, symbolic constants are declared separately from their fluent counterparts.

For example, consider the `carrying` fluent declared in terms of `robotAt(R,(X,Y),T)` and `shelfAt(S,(X,Y),T)` fluents rather than symbolic constants `robot(R)` and `shelf(S)`. In order to ground the term, all variables must be replaced with values. Although the fluents contain values `(R,S)`, they also introduce irrelevant variables `X,Y`, cause confusion with the time step value `T` occurring in all three fluents, and possibly overlap with intentional, well-formed rules or constraints.

Domain Independent Axioms and Choice Rules

Given a starting state, commonsense law of inertia states that the value of a fluent remains the same from one time step to the next, unless it is affected by an action or forced to change by another rule. Using choice rules, this is represented as

```
{val of fluent(T+1)} :- val of fluent(T)
```

where $\{\ldots\}$ denotes the set of choices, and the fluent has the option to keep its previous value. The choice rule allows

```
% commonsense law of inertia
{robotAt(R,(X,Y),T+1)} :-
    robotAt(R,(X,Y),T), T=0..m-1.

% uniqueness and existence of value constraints
:- not {robotAt(R,(X,Y),T)}=1, robot(R), T=0..m.

% actions are exogenous
{move(R,(DX,DY),T)} :-
    robot(R), DX=-1..1, DY=-1..1, T=0..m-1.
```

Figure 2: Domain independent axioms

this flexibility, where the fluent can take on a *default* value in the absence of any other rules that influence the fluent. For example, the position of a robot should remain the same from one time step to the next, unless it is affected by a move action.

To enforce uniqueness and existence of fluents, we use choice rules to declare that there can only exist exactly one choice for each fluent, at each time step. This is expressed as `1{fluent(T)}1` where the leftmost `1` denotes that *at least* one element must be chosen, and the rightmost `1` denotes that *at most* one element must be chosen. Constraints of this form are termed *cardinality constraints*, where the number of choices are bounded. In this case, the cardinality constraint ensures that there exists exactly one choice for each fluent at each time step and is equivalently declared as `{fluent(T)}=1`.

Note that on its own, the law of inertia does not guarantee that a robot's position will exist in the next time step since there is no lower bound on the choice rule. With the addition of the cardinality constraint, the law of inertia is enforced, and a robot *must* stay in its previous position unless moved.

Expressing this as a constraint, any solution that satisfies the negated rule `not {fluent(T)}=1`, violates the constraint, and is removed from the program's search space. This double negation semantic is referred to as *negation as failure*. The intuition is that a rule is assumed false, unless there is evidence that it is true. That is, solutions are assumed not to violate a constraint, unless it is proven to violate the constraint. In our scenario, solutions where a robot does not have a unique position at each time step is eliminated as an answer set, effectively ensuring that plans will have robots placed at unique positions at each time step.

The exogenous nature of actions can also be declared using choice rules. Together with the ground terms of the starting state, the election of actions generate the program's search space with all possible sequences of actions that can occur in the world, as well as the resulting state of the world after each action. For the warehouse, this axiom that declares actions to be exogenous essentially populates the universe of warehouse states from the initial warehouse state by generating the set of all possible sequence of actions.

Consider Figure 2 where the move action is declared as exogenous using an unbounded choice rule. The expression also specifies that `R` is ground via `robot(R)` and variables `DX,DY` ranges over values (-1,0,1). In a single time step, there are 9 possible moves for each robot, resulting in 9!

```
:- deliver(R,(O,P,U),T), Q<U,
    inventoryOn(S,(P,Q),T), lineItem(O,(P,U),T).
:- deliver(R,(O,P,Q),T), U<Q,
    inventoryOn(S,(P,Q),T), lineItem(O,(P,U),T).

{deliver(R,(O,P,D),T)} :-
    robot(R), order(O,P,U), D=1..U, T=0..m-1.
```

Figure 3: Deliver action

```
fulfillable(F,T) :- T=0..m,
    F=#count{1,O,P: lineItem(O,(P,U),T), U>0}.
:- not fulfillable(0,m).

makespan(T) :- fulfillable(F,T), F>0.
#minimize{1,T: makespan(T)}.
```

Figure 4: Objective function

sequences of moves to add to the program's search space. While grounding this rule is simple, it can be challenging to ground more complex actions like `deliver`.

Effects and Preconditions of Actions

The `deliver` action is the most challenging to represent, with its subtle preconditions and nuanced effect on the `inventoryOn` and `lineItem` fluents. For the sake of completeness, we will briefly touch on the other actions before diving into the `deliver` action.

Move This action affects positional fluents, and its effect on the `robotAt` is declared recursively. Any solutions where `|DX|=|DY|` are diagonal or stationary, and removed from the search space. We also disallow two robots to swap positions.

Shelves can be relocated by moving robots carrying shelves, and this is expressed recursively with the added conditions on the `carrying` fluent.

Other conflicting effects can be represented as state constraints, such as ensuring that the destination of a move is within the grid. This is represented as constraint that robots cannot be positioned on a non-node. Cardinality constraints are used to disallow two or more robots nor two or more shelves to occupy the same node and the same time. These rules also covers the cases where the effect of the move action may land either object on a node occupied by its like.

Pick up & Put down These two actions affect the `carrying(B,(R,S),T)` fluent, and their effect can be declared recursively. The pick up action sets the carrying state to true for `(R,S)` provided that the is located at the same node, and `R` is not presently carrying a shelf. Conversely, the put down action sets the carrying state to false for `(R,S)`, provided that `R` is presently carrying `S`, and they are not located on a highway node.

Deliver The fluents `inventoryOn(S,(P,Q),T)` and `lineItem(O,(P,U),T)` are affected by the `deliver(R,(O,P,D),T)` action, whereby the inventory quantity `Q`, and line item units `U`, are decreased by `D` number of products delivered. This effect is expressed recursively for both fluents. To ensure that the remaing inventory `(Q-D)` and fulfillable line items `(U-D)` do not fall below zero after the deliver action, the intuition is that delivered units `D` must be the lesser of the two values:

1. if `Q>=U`, then `D=U` to prevent *over delivery*
2. if `U>=Q`, then `D=Q` to prevent *owing inventory*

Consider the rule that declares the deliver action as exogenous. Grounding variables `R,O,P,T` is trivial, but the domain of `D` is not evident. The purpose of deliveries is to fulfill all line items of orders, and thus the lower bound of `D` is set as 1 and the upper bound is the *initial* number of units ordered. This is not necessarily the value of `U` in `lineItem(O,(P,U),T)`, that represents the *remaining* number of units fulfillable at time step `T`.

Recall the symbolic constant `order{O,P,U}` in Figure 1. This represents the initial state of the order line item, and as seen in Figure 3, it becomes incredible useful in grounding the deliver action.

Figure 3 also illustrates the constraints on `D` is expressed as a negated form of conditions 1 and 2 listed above. More literally:

1'. *throw out plans that over deliver* - remove solutions with `D=U` when `Q<U` since constraint is violated

2'. *throw out plans that owe inventory* - remove solutions with `D=Q` when `U<Q` since constraint is violated

Finally, we disallow deliveries to mismatched order picking stations, and ensure that robots are carrying the shelf that stocks the product in the order line item being delivered.

Goal State and Objective Function

The goal state is defined as a warehouse with all orders fulfilled, that is, all line items in the final state of the scenario must have no remaining units to fulfill. Expressed as a constraint,

```
:- not lineItem(O,(P,0),m), order(O,P,U).
```

where `m` is the maximum time step.

Equivalently, the goal state must have no line items that remaining fulfillable units. We define the `fulfillable(F,T)` aggregate to count `F` number of line items with non-zero fulfillable units at time step `T`. It follows that the `makespan(T)` value is expressed in terms of the `fulfillable` aggregate, the objective function is to minimize the makespan is demonstrated in Figure 4.

Results and Analysis

Using clingo, we solved the program initialized with the provided sample instances. In each case, the warehouse floor consisted of 16 nodes, 7 highways, and 2 robots. The maximum step size was set to 15 across all instances for consistency. The results are summarized in Table 1, where *Instance 0* denotes the example from the original problem description.

Table 1: Clingo Results

Instance	Stations	Shelves	Products	Orders	Optimal Makespan	Solve Time (s)
0	2	6	4	3	13	6.303
1	2	6	4	3	13	5.881
2	2	5	3	2	11	2.842
3	1	6	4	2	7	0.832
4	2	6	2	3	10	1.094
5	1	6	4	1	6	0.759

```
% ----- Alternate Program 1 -----
:- move(R,(DX,DY),T), robotAt(R,(X,Y),T),
    robotAt(RR,(X+DX,Y+DY),T), robotAt(RR,(X,Y),T+1).
% ----- Replaced by:
% :- move(R,(DX,DY),T), robotAt(R,(X,Y),T),
%     move(RR,(-DX,-DY),T), robotAt(RR,(X+DX,Y+DY),T).

% ----- Alternate Program 2 -----
inventoryOn(S,(P,Q-D),T+1) :- inventoryOn(S,(P,Q),T),
    deliver(R,(O,P,D),T), order(O,PS),
    robotAt(R,(X,Y),T), pickingStation(PS,(X,Y)).
% ----- Replaced by:
% inventoryOn(S,(P,Q-D),T+1) :-
%     inventoryOn(S,(P,Q),T), deliver(R,(O,P,D),T).
% :- deliver(R,(O,P,D),T), order(O,PS),
%     robotAt(R,(X,Y),T), not pickingStation(PS,(X,Y)).
```

Figure 5: Alternate programs

As the number of stations, shelves, products, and orders increase, the optimal makespan and solve time increases. This is expected since the program's search space increases with the number of objects and actions, and the number of possible sequences of actions increases exponentially with the number of time steps.

We can also review the performance of our program by replacing constraints with different representations that yield an optimal model of equivalent size. This provides insights on how changes in the representation of constraints may affect the program's search space, how effectively the solver can reduce the search space with a different representation, and the effects on solve time to finding an optimal model.

In the alternate programs illustrated in figure 5, we use initialize with Instance 0 and maximum step size 20. The original program takes 12.379s to find an optimal plan with a makespan of 13.

Alternate program 1 The constraint for swapping positions is modified from having 1 action and 3 fluents, to 2 actions and 2 fluents. It takes clingo 35.649s to find an optimal plan with a makespan of 13. This is a significant increase in solve time, and the program is not as effective in reducing the search space.

Alternate program 2 The delivery effect on inventory is modified from having the designated picking station condition inline with the recursive definition, to expressing this as a separate constraint. It takes clingo 19.050s to find an optimal plan with a makespan of 13. Here we see that having constraints as part of the recursive definition is more effective in reducing the search space.

Conclusion

This project has been a great learning experience in applying answer set programming to a real-world scenario. Through trialing various representations, it has put into context theories and formalisms learned in lectures, and helped connect the dots between abstract concepts such as the implication of exogenous actions on the program search space. The project has been instrumental in gaining a deeper understanding of knowledge representation and reasoning in practice, and answer set programming and its applications.

Opportunities for Future Work

In the original scenario description, there is mention of permitting robots to perform multiple concurrent actions at each time step. This would involve modifying the program to generate only serializable plans, where actions that are scheduled for the same time step can be executed in any order, without affecting the result. It would be interesting to explore the changes to satisfability and optimality in the modified program.

References

Anger, C.; Konczak, K.; Linke, T.; and Schaub, T. 2005. A glimpse of answer set programming. *KI* 19:12–.

Gebser, M.; Ostrowski, M.; and Schaub, T. 2009. Constraint answer set solving. 235–249.

Kaminski, R., and Schaub, T. 2022. On the foundations of grounding in answer set programming.

Kaufmann, B.; Leone, N.; Perri, S.; and Schaub, T. 2016. Grounding and solving in answer set programming. *AI Magazine* 37:25–32.

Kowalski, R., and Sadri, F. 2000. Reconciling the event calculus with the situation calculus. *The Journal of Logic Programming* 31.

Lee, J., and Palla, R. 2012. Reformulating the situation calculus and the event calculus in the general theory of stable models and in answer set programming. *Journal of Artificial Intelligence Research* 43:571–620.

Lifschitz, V. 2016. Answer sets and the language of answer set programming. *AI Magazine* 37:7–11.