CSE 579 Individual Project Report
Automated Warehouse Scenario

James Butcher
ASU ID: 1220960052
jbutche6@asu.edu

Problem Statement

This project is based on a problem from the 2019 ASP Challenge. It is a simplification of the real-world application of using robots to transport items in a warehouse. The setting is a rectangular grid of cells, some of which contain shelves, robots, and/or picking stations. The shelves contain products meant to fulfill customer orders. The goal is to have the robots move to the shelves containing the order items, pick up the shelves, bring them to the picking stations, and drop off the items to fulfill all the orders in as little time as possible.

Here is the problem in more detail: The warehouse is represented as a rectangular grid of cells or *nodes*, each indexed by an X and Y location. One or more *robots* may *move* from cell to cell in the grid, either up, down, left, or right, but not diagonally. While moving, robots must not collide with each other. There are also *shelves* at various cells in the warehouse. Shelves may contain certain *products* in various quantities. There are outstanding *orders* to be fulfilled, consisting of certain quantities of some of these products. Orders are to be fulfilled at certain *picking stations*, which are permanently located at certain cells in the warehouse. Robots can move underneath (that is, move into the same cell), *pick up*, move, and *put down* shelves, one at a time. A robot that is carrying a shelf cannot move into a cell containing another shelf. Certain cells in the warehouse are also designated as *highways*, which are cells where no shelves are allowed to be put down. These highways are meant to be reserved pathways for robots to move and carry shelves freely. If a robot carries a shelf containing the correct products to fulfill an order to the order's corresponding picking station, the robot can *deliver* those items at the picking station, thus fulfilling that part of the order. Each robot may carry out at most one action per time step. The goal is to fulfill all the orders in as little time as possible.

Project Background

Everything that follows on this project's background I learned from Arizona State University's CSE 579 - Knowledge Representation and Reasoning course on Coursera created by Dr. Joohyung Lee.

This project is a Knowledge Representation and Reasoning (KRR) problem. The challenge is to represent common-sense knowledge we humans have about time, space, movement, and other physical facts (such as how objects cannot pass through one another) in a form that a computer can reason about. If successful, then the computer can find optimal solutions much faster than a human can. In this case, it can find an optimal sequence of actions for robots to deliver products to picking stations in a warehouse.

The task was attempted using Answer Set Programming (ASP). ASP is a form of declarative programming useful for knowledge-intensive applications such as this (Lifschitz 2008).

Declarative programming differs from traditional programming by describing *what* the program must accomplish rather than *how* to accomplish it. In this project, I describe the warehouse scenario and the goal, and let the solver search for solutions that satisfy everything that has been declared. In ASP, these solutions are called *stable models.*

The programming methodology for ASP is often called "Generate, Define, Test." First, you generate a set of potential solutions. For example, you may state that a robot may deliver any quantity of any product at any time T. Then, in the test step, you write constraints that prune out all "bad" solutions. For example, you may state that a robot cannot make a delivery if it is not at the same location as the correct picking station, or, state that two shelves cannot be in the same location at the same time. The "define" step consists of defining auxiliary predicates, which can be useful for complicated programs (Lifschitz 2008).

ASP can also handle problems having to do with states and actions over time, called transition systems. The warehouse scenario is one such problem. In a transition system, states of the world that can change their value over time are called fluents. For example, the state representing whether a robot is carrying a shelf at a given time T is a fluent. At one time step, the robot may be carrying a shelf, while at another time step it may not.

There are several things that must be explicitly encoded in ASP in order to ensure a transition system makes sense. One such thing is called the common sense law of inertia. That is, making sure that states of the world do not change over time when no action is taken (Shanahan 2004). For example, if the robot is carrying a shelf at time T=1 and the robot takes no action, then it should still be carrying the shelf at time T=2.

Another common sense concept that must be encoded is the uniqueness and existence of fluent values. This means that, for example, there must be one and only one value for whether a robot is carrying a shelf at all time steps. This constraint prohibits the fact from being either: "both true and false at the same time" or "neither true nor false at the same time." The same goes for non-boolean fluents with many possible values. For example, a robot must be at exactly one location at each time step.

It is also necessary to encode the concept that actions are exogenous, meaning that they may or may not occur at any given time step. This gives the ASP solver free rein to search for any combination of actions that satisfy all the constraints.

With this domain-independent foundation in place, the next task is to define the specific aspects of the problem domain. All of the words I italicized in the problem statement section above represent the key *objects* and *actions* to be represented. For example, the statement: [object(robot,1).] represents an object we may call "Robot number 1."

For this project I used Clingo, an ASP solver language. Throughout this paper, I will include snippets of Clingo code enclosed in square brackets, as in the above paragraph. To learn more about the Clingo language, visit https://potassco.org/clingo/.

The Approach

To tackle this complex problem I used a small, step-by-step approach. I started with highly simplified scenarios, got them to work on a basic level first, and then slowly built up the system to represent more complicated states, making sure everything worked each step of the way.

The general procedure was to write some Clingo code, execute the file on the command line, and examine the output. Assuming there were no syntax errors, there were three output possibilities:

1. UNSATISFIABLE - this means that Clingo could not find a stable model for the system.
2. SATISFIABLE + (correct states and actions) - this means that Clingo found the correct solution.
3. SATISFIABLE + (incorrect states and/or actions) - this means that I did not represent the system correctly.

I divided the work into four main parts, one for each action: move, pickup, putdown, and deliver.

For each one, I first wrote out the domain-independent axioms (common sense law of inertia, uniqueness and existence of fluent values, etc.) because these were both easy to write and necessary before testing the actions at even a basic level. This approach worked well. I created simplified test cases for each action step and let each action sub-task guide me in the creation of state and action constraints.

The problem description included sample Clingo code to build off of. The solution was intended to use this code as a model. The initial states of the system were provided in this form:

[init(object(robot,1), value(at, pair(4,3))).],

and actions were provided in this form:

[occurs(object(robot,1), move (-1,0),1).].

The first challenge was grasping the structure of these large atoms. What was the purpose of the "occurs" predicate? Why did an object and a value need to be grouped together inside parentheses? It took a while playing around with smaller, more manageable transition systems before I realized it was needed to link the object and the action together into a single predicate. Since this problem deals with many instances of objects doing multiple things at different times, the statements keeping track of them necessarily become large and complicated.

With the purpose of building up my own grasp of the large atoms and getting my own custom ones to work, I made a highly simplified version of the warehouse problem without large atoms. I had to make sure I could get that to work first. Using a 3x3 grid, an "at" predicate with arity 3, and a simple move action, I succeeded in getting Clingo to find all six stable models for how to move the robot from pair(1,1) to pair(3,3) in four time steps.

I realized at once that I needed to create a large, flexible predicate to represent the fluents, such as the location of the robots at any time T. I first considered using "occurs" to denote everything: both actions and fluents. Although it would work, it wouldn't make much sense. "Occur" sounds more like an active change. An object being at a location would be better described as a "state" rather than an occurrence. And so I decided to reserve the predicate "occurs" for actions and introduce a new predicate, "state," for fluents. Thus the move action definition looked like this:

[state(object(robot,R),value(at,pair(X+DX,Y+DY)), T+1) :- occurs(object(robot, R), move(direction(DX, DY)), T), state(object(robot, R), value(at, pair(X, Y)), T).]

(This statement means "If robot R is at location X, Y at time T and it also moves in the direction DX, DY at time T, then at time T+1 it will be at location X+DX, Y+DY.)

This naming convention had the added advantage of making the Clingo outputs easier to read. I could easily tell whether an atom was a fluent or an action by looking at its main predicate.

Once I got the robot to move across the grid using my working formalism, the next challenge was to get two robots to move past each other without colliding. I achieved this by first adding a constraint prohibiting two different robots from occupying the same X, Y pair at the same time. This didn't prevent two robots from swapping locations from one time step to the next, though. So then I added four constraints prohibiting up, down, left, and right movement into a location occupied by another robot at the same time step. This worked. I had now solved the movement problem for multiple robots in the warehouse. See the appendix for the full code.

The next task was to solve the pickup action. My first attempt was based on the idea that a robot would explicitly carry a specific shelf, like this:

```
[state(object(robot,R),value(is_carrying_shelf,object(
shelf, S)), T).]
```

This demanded lots of complicated supporting code, such as the need for a "dummy shelf" (shelf 0) to take the place of the shelf object whenever a robot was not carrying a shelf.

I found a much simpler solution by instead just using a boolean object (either true or false) to indicate whether a robot was carrying a shelf or not. So initially a robot would be in the state of not carrying a shelf, like so:

[state(object(robot, R), value(carrying, **false**), 0).]

And so the pickup action looked like this:

[state(object(robot, R), value(carrying, **true**), T+1) :-
occurs(object(robot, R), pickup, T).]

But how does the system know which shelf the robot is carrying? How does it know to move the shelf to wherever the robot moves? The answer is to use a state constraint. I simply wrote a line of code dictating that whenever a robot with "carrying" equal to true moves from X1,Y1 to X2,Y2, then the shelf at the same location must move also:

```
[state(object(shelf, S), value(at, pair(X2, Y2)), T+1) :-
state(object(robot, R), value(carrying, true), T),
state(object(robot, R), value(at, pair(X1, Y1)), T),
state(object(shelf, S), value(at, pair(X1, Y1)), T),
state(object(robot, R), value(at, pair(X2, Y2)), T+1).]
```

It wasn't as easy as that, however. I needed to add some additional constraints: prevent a robot from picking up a shelf if there was no shelf there, prevent a robot from picking up a shelf if it was already carrying one, and prevent two shelves from being at the same location at the same time:

```
[ :- state(object(shelf, S1), value(at, pair(X, Y)), T),
state(object(shelf, S2), value(at, pair(X, Y)), T),
S1 != S2. ]
```

This last constraint (above) nicely took care of ensuring that a robot carrying a shelf does not move into a location containing another shelf: for if it did, then at the next time step there would be two shelves at the same location and it would violate the constraint. This rule therefore made those invalid movements *redundant*, and they could therefore be omitted.

At this point, since the robots were now capable of multiple actions, I needed to begin adding constraints to prevent *concurrent actions* such as moving and picking up a shelf at the same time:

```
[ :- occurs(object(robot, R), move(direction(DX, DY)),
T), occurs(object(robot, R), pickup, T). ]
```

It was a simple matter adding more of these constraints after adding the putdown and deliver actions: (a robot cannot move and put down a shelf at the same time, a robot cannot move and deliver at the same time, and so on.)

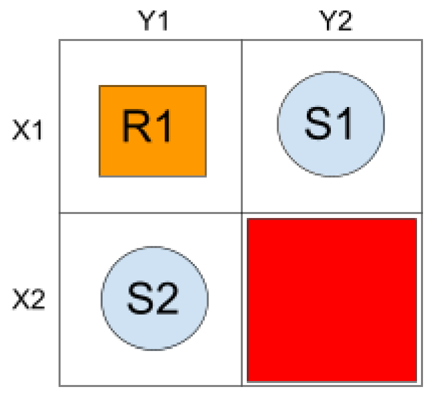
I verified this stage in the project using another simplified warehouse scenario. This time, I made one robot move one shelf to a goal location. Then, I made two robots move one shelf each to two different goal locations, while making sure there were no collisions.

After that, it was a simple matter to add the putdown action. Basically, I just implemented the opposite of the pickup action: it changed the robot's carrying state from true to false. The only other constraint to add was to prohibit robots from putting shelves down on locations designated as highways.

Unfortunately, I was unable to solve the full problem in time. I could not get the deliver action to work properly beyond simple test cases.

Main Results and Analysis

At the time of this writing, my solution was capable of solving goal states involving moving shelves to goal locations, including cases where other shelves needed to be moved out of the way. The following example shows such a case.



The above scenario depicts a 2x2 warehouse with robot R1 initially at location (1, 1), a shelf S1 at (1, 2), another shelf S2 at (2, 1), and the red square at location (2, 2) designating the cell as a highway. The goal was for shelf S1 to be at location (2, 1) at time m:

```
[ :- not state(object(shelf, 1), value(at, pair(2, 1)), m). ]
```

Intuitively, one knows that the robot must first move S2 out of the way. Since shelves cannot be placed on highway cells, it is clear that S2 must be put down on cell (1, 1). Then, the robot should pick up S1, and, since shelves cannot pass through each other, the robot should move down and then left via the highway cell, bringing the shelf S1 to cell (2, 1), thereby fulfilling the goal. This is a total of eight actions. And this is exactly what happens when I run the code with m=8:

Input command:

```
clingo checkpoint1.lp input1.lp -c m=8 0
```

Output (modified for clarity):

```
Answer: 1
occurs(object(robot,1),move(direction(1,0)),0)
occurs(object(robot,1),pickup,1)
occurs(object(robot,1),move(direction(-1,0)),2)
occurs(object(robot,1),putdown,3)
occurs(object(robot,1),move(direction(0,1)),4)
occurs(object(robot,1),pickup,5)
occurs(object(robot,1),move(direction(1,0)),6)
occurs(object(robot,1),move(direction(0,-1)),7)
SATISFIABLE
Models      : 1
```

Finally, for the deliver action, I succeeded with small test cases similar to the above for delivering products to fulfill orders, but I failed to fix some key issues that prevented my solution from scaling up to the full-size test cases. One was a case where one robot at one picking station would deliver part of another product's order held by another robot at another picking station, and vice versa.

Another problem occurred when I made a constraint enforcing a shelf containing a product to be at the correct picking station. The system became unsatisfiable when I put quantities of the same product on different shelves. Apparently, the solver seemed to want *all* of the shelves containing that same product for delivery to be at the picking station at the same time.

I think the main problem behind both of these issues was ambiguity between products and shelves and their locations during deliveries. I could not enforce the idea that *only the products on the shelf at the picking station where the delivering robot is, and only the ones that fulfill the order in question* should be deducted from the shelf. It seems likely to me that the solution lies in explicitly tying a robot to the specific shelf it is carrying instead of simply declaring whether the robot is carrying a shelf or not. This was the idea I had abandoned while working on the pickup action. Perhaps it would have worked if I had kept going down that route.

Conclusion

Using the knowledge I gained from CSE 579 - Knowledge Representation and Reasoning, I was able to implement three out of four of the actions in the automated warehouse scenario: move, pickup, and putdown. Although I was unable to solve the fourth action: deliver, I am confident that I could have solved it with more time and hard work.

The hardest part of the project, especially for the deliver action, was keeping all of the variables coordinated. During a deliver action, there needed to be a robot, a product, an order, a shelf, a picking station, a time step, and an X and Y location all synchronized together. This juggling act overwhelmed me. If I had to do this project again, I would concentrate on first making more helper predicates to help coordinate sub-tasks, such as associating a robot with a specific shelf it is carrying. In other words, I would focus more on the "Define" step in the "Generate, Define, Test" methodology.

Opportunities for Future Work

One intriguing area I would like to learn about is the application of ASP to physics. Thinking about having to explicitly encode common-sense knowledge such as how objects cannot pass through each other brought to mind the question: what if they can? Many of the principles of quantum mechanics and relativity famously defy common sense.

It would be fun to experiment with ASP systems that ignore common sense notions and see where they lead.

References

Lifschitz, V. 2008. What is Answer Set Programming? In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence*. Menlo Park, California: AAAI Press. https://www.aaai.org/Papers/AAAI/2008/AAAI08-270.pdf.

Shanahan, M. 2004. The Frame Problem. *The Stanford Encyclopedia of Philosophy,* Spring 2016 edition. https://plato.stanford.edu/entries/frame-problem/

Appendix

```
% James Butcher
% Warehouse 2.0
% 2-26-2021

% ---------------------------------------------------------------------------
% Unpack Input
% ---------------------------------------------------------------------------

object(O, V) :- init(object(O, V), value(_, _)).
value(A, V) :- init(object(_, _), value(A, V)).

state(object(O, V1), value(A, V2), 0) :- init(object(O, V1), value(A, V2)).


% ---------------------------------------------------------------------------
% Goals
% ---------------------------------------------------------------------------

% Fulfill all orders
:- not state(object(order, D), value(line, pair(I, 0)), m), init(object(order, D), value(line, pair(I, U))).


% ---------------------------------------------------------------------------
% Sort and Object Declaration
% ---------------------------------------------------------------------------

direction(0, 1; 0, -1; 1, 0; -1, 0).
boolean(true, false).


% ---------------------------------------------------------------------------
% Other Object Initialization
% ---------------------------------------------------------------------------

% Robots start not carrying a shelf
state(object(robot, R), value(carrying, false), 0) :- object(robot, R).

% Statement indicating if two objects are at the same location
same_location(object(O1, V1), object(O2, V2), T) :- state(object(O1, V1), value(at, pair(X, Y)), T), state(object(O2, V2),
value(at, pair(X, Y)), T).

% *** An example of what I should have done more of - this simple predicate greatly simplified the putdown constraint
% Indicate immovable object locations for easy location matching
is_highway(X, Y) :- init(object(highway, H), value(at, pair(X, Y))).


% ---------------------------------------------------------------------------
% State Constraints
% ----------------------------------------------------------------------------

% If a robot is carrying a shelf, then the shelf must be at the same location as the robot in the next time step if the robot
moves
state(object(shelf, S), value(at, pair(X2, Y2)), T+1) :- state(object(robot, R), value(carrying, true), T),
                                                         state(object(robot, R), value(at, pair(X1, Y1)), T),
                                                         state(object(shelf, S), value(at, pair(X1, Y1)), T),
                                                         state(object(robot, R), value(at, pair(X2, Y2)), T+1).

% No two robots can be at the same location at the same time
:- state(object(robot, R1), value(at, pair(X, Y)), T), state(object(robot, R2), value(at, pair(X, Y)), T), R1 != R2.

% Two shelves cannot be in the same location at the same time
:- state(object(shelf, S1), value(at, pair(X, Y)), T), state(object(shelf, S2), value(at, pair(X, Y)), T), S1 != S2.

% A robot cannot move and pick up a shelf at the same time
:- occurs(object(robot, R), move(direction(DX, DY)), T), occurs(object(robot, R), pickup, T).

% A robot cannot move and put down a shelf at the same time
:- occurs(object(robot, R), move(direction(DX, DY)), T), occurs(object(robot, R), putdown, T).

% A robot cannot move and deliver at the same time
:- occurs(object(robot, R), move(direction(DX, DY)), T), occurs(object(robot, R), deliver(O, I, Q), T).

% A robot cannot move and put down a shelf at the same time
:- occurs(object(robot, R), putdown, T), occurs(object(robot, R), deliver(O, I, Q), T).


% ----------------------------------------------------------------------------
% Actions
% ----------------------------------------------------------------------------

% Move action
%-----------
state(object(robot, R), value(at, pair(X + DX, Y + DY)), T+1) :- occurs(object(robot, R), move(direction(DX, DY)), T),
                                                                 state(object(robot, R), value(at, pair(X, Y)), T).

% Cannot move out of bounds, i.e, to a location not declared in the beginning
:- occurs(object(robot, R), move(direction(DX, DY)), T), state(object(robot, R), value(at, pair(X, Y)), T),
   not 1{value(at, pair(X + DX, Y + DY))}.

% Avoid collisions with other robots - also takes care of making actions serializable
:- state(object(robot, R1), value(at, pair(X1, Y)), T), state(object(robot, R2), value(at, pair(X2, Y)), T), (X1 - X2) == 1,
   occurs(object(robot, R1), move(direction(-1, 0)), T).
:- state(object(robot, R1), value(at, pair(X1, Y)), T), state(object(robot, R2), value(at, pair(X2, Y)), T), (X1 - X2) == -1,
   occurs(object(robot, R1), move(direction(1, 0)), T).
:- state(object(robot, R1), value(at, pair(X, Y1)), T), state(object(robot, R2), value(at, pair(X, Y2)), T), (Y1 - Y2) == 1,
   occurs(object(robot, R1), move(direction(0, -1)), T).
:- state(object(robot, R1), value(at, pair(X, Y1)), T), state(object(robot, R2), value(at, pair(X, Y2)), T), (Y1 - Y2) == -1,
   occurs(object(robot, R1), move(direction(0, 1)), T).
```

```
% Pickup action
%--------------
state(object(robot, R), value(carrying, true), T+1) :- occurs(object(robot, R), pickup, T).

% Must be a shelf at the robot's location
:- occurs(object(robot, R), pickup, T), not 1{same_location(object(robot, R), object(shelf, S), T) : object(shelf, S)}1.

% Cannot pick up a shelf if the robot is already carrying a shelf
:- occurs(object(robot, R), pickup, T), state(object(robot, R), value(carrying, true), T).

% Putdown action
%---------------
state(object(robot, R), value(carrying, false), T+1) :- occurs(object(robot, R), putdown, T).

% Cannot put down a shelf if the robot isn't carrying a shelf
:- occurs(object(robot, R), putdown, T), state(object(robot, R), value(carrying, false), T).

% Cannot put down a shelf on a highway
:- occurs(object(robot, R), putdown, T), state(object(robot, R), value(at, pair(X, Y)), T), is_highway(X, Y).

% Deliver action
%---------------
% Subtract delivered quantity of product from shelf
state(object(product, I), value(on, pair(S, U - Q)), T+1) :- occurs(object(robot, R), deliver(O, I, Q), T),
                                                             state(object(product, I), value(on, pair(S, U)), T).
% Subtract delivered quantity of product from order
state(object(order, O), value(line, pair(I, U - Q)), T+1) :- occurs(object(robot, R), deliver(O, I, Q), T),
                                                            state(object(order, O), value(line, pair(I, U)), T).

% *** This part was the main source of difficulty - doesn't work properly - in process of changing at time of writing ***
% Cannot deliver if robot is not at same location as correct pickingStation
:- occurs(object(robot, R), deliver(O, I, Q), T), state(object(order, O), value(pickingStation, N), T),
   not same_location(object(robot, R), object(pickingStation, N), T).

% Cannot deliver if shelf is not at same location as correct pickingStation
:- occurs(object(robot, R), deliver(O, I, Q), T), state(object(product, I), value(on, pair(S, U)), T),
   state(object(order, O), value(pickingStation, N), T), same_location(object, robot, R), object(shelf, S),
   not 1{same_location(object(shelf, S), object(pickingStation, N), T)}.


% ----------------------------------------------------------------------------
% Domain-Independent Axioms
% ----------------------------------------------------------------------------

% Actions are exogenous
{occurs(object(robot, R), move(direction(DX, DY)), T) : direction(DX, DY)} :- object(robot, R), T = 0..m-1.
{occurs(object(robot, R), pickup, T)} :- object(robot, R), T = 0..m-1.
{occurs(object(robot, R), putdown, T)} :- object(robot, R), T = 0..m-1.
{occurs(object(robot, R), deliver(O, I, U), T) : object(order, O), object(product, I), U = 1..4} :- object(robot, R), T = 0..m-1.
```

```
% Existence and uniqueness of fluent values
:- not 1{state(object(robot, R), value(carrying, B), T)}1, object(robot, R), T = 1..m.
:- not 1{state(object(robot, R), value(at, pair(X, Y)), T)}1, object(robot, R), T = 1..m.
:- not 1{state(object(shelf, S), value(at, pair(X, Y)), T)}1, object(shelf, S), T = 1..m.
:- not 1{state(object(pickingStation, N), value(at, pair(X, Y)), T)}1, object(pickingStation, N), T =  1..m.
:- not 1{state(object(product, I), value(on, pair(S, U)), T)}1, object(product, I), object(shelf, S), T = 1..m.
:- not 1{state(object(order, O), value(line, pair(I, U)), T)}1, object(order, O), T = 1..m.
:- not 1{state(object(order, O), value(pickingStation, N), T)}1, object(order, O), T = 1..m.

% Common sense law of inertia
{state(object(robot, R), value(carrying, B), T+1)} :- state(object(robot, R), value(carrying, B), T), T = 0..m-1.
{state(object(robot, R), value(at, pair(X, Y)), T+1)}  :- state(object(robot, R), value(at, pair(X, Y)), T), T = 0..m-1.
{state(object(shelf, S), value(at, pair(X, Y)), T+1)}  :- state(object(shelf, S), value(at, pair(X, Y)), T), T = 0..m-1.
{state(object(pickingStation, N), value(at, pair(X, Y)), T+1)} :- state(object(pickingStation, N), value(at, pair(X, Y)), T), T = 
0..m-1.
{state(object(product, I), value(on, pair(S, U)), T+1)} :- state(object(product, I), value(on, pair(S, U)), T), T = 0..m-1.
{state(object(order, O), value(line, pair(I, U)), T+1)} :- state(object(order, O), value(line, pair(I, U)), T), T = 0..m-1.
{state(object(order, O), value(pickingStation, N), T+1)} :- state(object(order, O), value(pickingStation, N), T), T = 0..m-1.

#show state/3.
#show occurs/3.
```