Project Milestone 4
Automated Warehouse Scenario

Aditi Shashank Joshi
ASU ID: 1222838916
ajoshi64@asu.edu

Problem Statement

This project is based on a problem statement from 2019 ASP (Answer Set Programming) challenge. The main purpose of this project is to deliver the packages by robots located inside warehouses. Robots need to pick up the shelves which contains the products which needs to be delivered, make it available at the warehouse, and then later drop it at the picking station. The parcel should be available as soon as possible, based on the given orders. This represents a real time warehouse scenario.

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.

This project is a Knowledge Representation and Reasoning (KRR) problem. The challenge is to represent commonsense 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.

Declarative programming differs from traditional programming by describing *what* the program must accomplish rather than *how* to accomplish it. This project describes 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 ware-

house 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 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 to ensure a transition system makes sense. One such thing is called the commonsense 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 commonsense 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 fluent 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 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 - Clingo could not find a stable model for the system.
2. SATISFIABLE + (correct states and actions) - Clingo found the correct solution.
3. SATISFIABLE + (incorrect states and/or actions) – representation of the system is wrong.

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

For each one, firstly the domain-independent axioms (common sense law of inertia, uniqueness, and existence of fluent values, etc.) were written because these were both easy to write and necessary before testing the actions at even a basic level. Simplified test cases were created for each action step and each action sub-task guided in the creation of state and action constraints.

The problem description included sample Clingo code to build off. 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 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) 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.

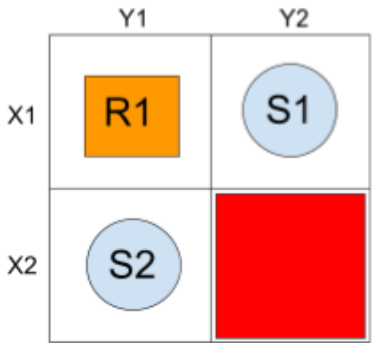
After that, it was a simple matter to add the putdown action. Basically, I just implemented the opposite of the pickup action: it changed the robot's carrying state from true to false. The only other constraint to add was to prohibit robots from putting shelves down on locations designated as highways.

Main Result and Analysis

Clingo's situations could be visualized using the asprilo tool. I created the scenario by defining the numerous actions and instances in strict accordance with the project description we were given. The stable models generated a list of all the activities taken by the robot that resulted in all the orders being completed. This was determined by storing the number of product-specific items in an atom named order. Every time the action deliver occurred; this atom would change. As a result, this atom was the primary determinant of the terminating condition. The results of running the clingo program might be converted into an asprilo-specific format, allowing viewing of the sequence of events. I was then able to validate all the limits that were supposed to be placed on the robot once this was in place. A robot, for example, would never place a shelf on the roadway or go under a shelf while carrying another shelf.

When I first tried to integrate the "deliver" functionality, I ran across several problems. The limits imposed were insufficient, and the software became inefficient, resulting in

an infinite loop. This needed to be rebuilt, so I added more conditions that allowed the robot to only perform one action at a given time stamp. Furthermore, two sorts of deliveries had to be created: one where the number of things supplied was less than the required order, and another where the number of products on the shelf was greater than the current order requirement. Adding limitations that limited the number of things provided to be greater than zero but less than the actual demand resulted in a terminating condition at the ideal time.



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).]

Warehouse scenario	Minimal Timesteps involved
1	13
2	11
3	7
4	10
5	6

Table 1. Minimal timesteps for solving all simple instanes.

My result shows the actions taken by the robots to execute the orders in the five instances and the time (steps) taken. To execute the actions the robots carried out movements according to the given constraints, such as robots should not collide, robots should not go through shelves when carrying a shelf, robots should stay within the rectangular grid, etc.

Conclusion

We were given a simplified version of several Amazon Warehouse Situations for this course assignment, and we were to write a program to solve these scenarios by satisfying all orders in the shortest amount of time feasible. I was able to design a solver that could be applied to these cases to develop a solution using my understanding of Answer Set Programming and Clingo.

Using the knowledge, I gained from CSE 579 - Knowledge Representation and Reasoning, I was able to implement four of the actions in the automated warehouse scenario: move, pickup, putdown and deliver.

The hardest part of the project, especially for the deliver action, was keeping all 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.

Overall, this project is one of the most practical, real world utilization of ASP and I have executed to the best of my ability.

Opportunities for Future Work

This project scenario is a simplified version of the Amazon warehouse. So, for future work, this problem can be developed further to represent the Amazon warehouse with more structural accuracy.

We can expand the grid world to account for moving warehouse employees. An additional “restock” action can be implemented which will restock the shelves with the necessary products when a shipment truck arrives at the warehouse. Consequently, we can add a “restock station” where the robots will carry the shelves which need to be restocked.

References

1. Nguyen, V., Obermeier, P., Son, T. C., Schaub, T., & Yeoh, W. (2017), *Generalized Target Assignment and Path Finding Using Answer Set Programming*, IJCAI (pp. 1216-1223). ijcai.org.
2. https://sites.sju.edu/plw/files/2019/06/slides.pdf
3. Marcello Balduccini. *Representing Constraint Satisfaction Problems in Answer Set Programming.*