CSE 505 Research Project

Zach Dobroff
SUNY Stony Brook

December 16, 2018

Introduction

Experimenting with robotic intra-logistics domains is a research paper (by Martin Gebser, Philipp Obermeier, Thomas Otto and Torsten Schaub University of Potsdam, Germany Orkunt Sabuncu TED University, Ankara, Turkey Van Nguyen and Tran Cao Son New Mexico State University, Las Cruces, USA) detailing how to use Answer Set Programming in order to experiment with automatizing warehouse operations using robots. The paper introduces a tool called asprilo (asprilo stands for Answer Set Programming for robotic intra-logistics), which is built in clingo and Python, designed to visualize, solve and benchmark given problems in this space.

Paper Summary

Defining the problem

The problems are generally defined as such:

- A **warehouse** is a two-dimensional grid of squares
- Each grid square is called a **node** and has a unique X,Y value
- A warehouse contains **robots**, **shelves**, **stations**, **storage areas** and **highways**
- Each of the above is one grid square in size
- Robots are mobile and carry out **orders**
- Robots can carry shelves
- Each time step, a robot can either **move**, **pick up a shelf**, **put down a shelf** or **deliver** a product to a station
- Only one robot may occupy a given square at a time
- One robot can share a square with one shelf (a robot carrying a shelf must go around other shelves)
- An order is a non-empty set of **order lines**, which are requests for a certain quantity of a certain product, delivered to a certain station
- An order is fulfilled if all order lines are fulfilled
- Shelves are stationary and contain **products**

- A storage areas is a grid square where shelves can be placed
- Only one shelf can occupy a given square at a time
- Stations act as a destination (a robot delivers a product to a station to fulfill an order)
- Highway grid squares are transit areas for travelling robots (idle robots should not stop on highways)
- Shelves cannot be placed on highways

Additionally, the paper specifies several problem constraints, called **domains**:

- Domain A
  - Product quantities are required at a station for an order to be completed
  - A single delivery can only fulfill a single order line
  - Multiple order lines require multiple deliveries
- Domain B
  - A product must be delivered to the station for the order to be completed, but in any quantity
  - Like Domain A, a single delivery can only fulfill a single order line
  - Like Domain A, multiple order lines require multiple deliveries
- Domain C
  - One delivery action can fulfill multiple order lines
- Domain M
  - A drastic simplification of the problem
  - Orders are fulfilled if a robot is under a shelf containing the desired product
  - Shelves only contain a single unique product
  - No deliveries are required
  - No pick up or put down actions are required
- Domains A^M, B^M and C^M
  - Only deal with singleton orders using the proper domain constraints
  - Shelves can only contain one type of product

Asprilo

The asprilo program is made up of several components, an instance generator, a checker and a visualizer. The generator takes a series of arguments and outputs an ASP file containing initial objects and their locations according to the inputted specifications. The generator supports three types of layouts, structured, randomized and customized. In the structured layout, stations and robots are initially placed in the upper and lower row, respectively. Shelves are placed in rectangular clusters reachable via surrounding highways. Additionally, the generator can create instances in batches via a Python script. The checker is where the main logic occurs: robots move, deliver products and orders are completed. The checker throws errors if orders cannot be completed or other problems occur. The visualizer is a Python program that shows an animation of the individual steps taken in the problem-solving process. A user can create instances using the visualizer instead of using the generator program, but only one at a time.

Implement, Verify and Extend

Installation

Asprilo requires clingo (clingo can be downloaded from Potassco's github), Python version 2.x, the clingo Python module (available through an Anaconda distribution, be sure to download a version with py27 in its name) and either a Linux or Mac operating system. Unfortunately, asprilo does not run well on Windows, this is because asprilo is written in Python 2.x using an imported clingo module. This clingo module is not compatible with Python 2.x on Windows, but a compatible version exists for Linux or Mac.

Problem Source

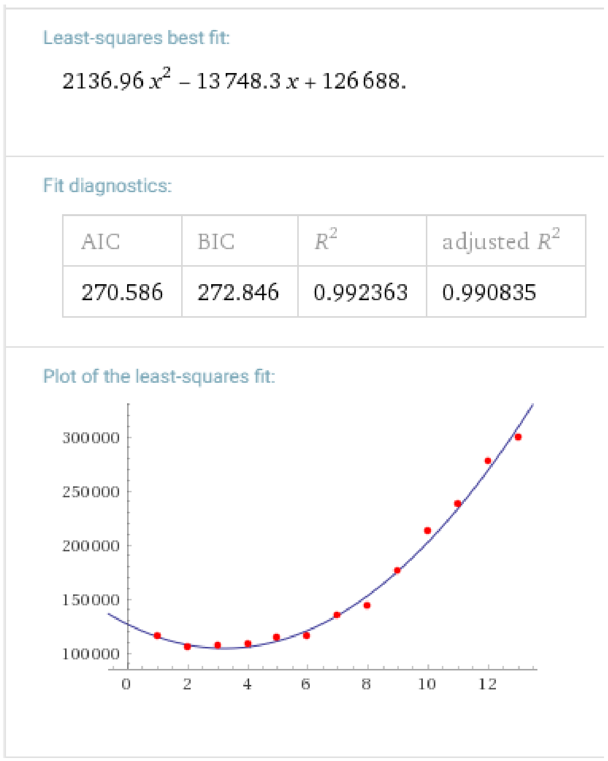
The paper has a table of example problem sets with a domain, size and number of robots, showing how long it took for that problem set to resolve. Some of the entries timed out and unfortunately, the paper does not say why. My initial thought was to classify timeouts as one of the following:

- Traffic jam
  - The robots jam up somewhere, creating gridlock which makes some orders impossible to fulfill
  - Too many objects, not enough space for the robots to move
- Poor path finding
  - May have some overlap with traffic jams
- Impossible objective
  - Maybe the robots cannot reach the stations or shelves
- Other

After running several tests of my own, I determined that the reason for the timeouts was due to the solving algorithm running in exponential time: $\mathrm{O(N^R)}$ where R is the number of robots in the instance.

Time Complexity

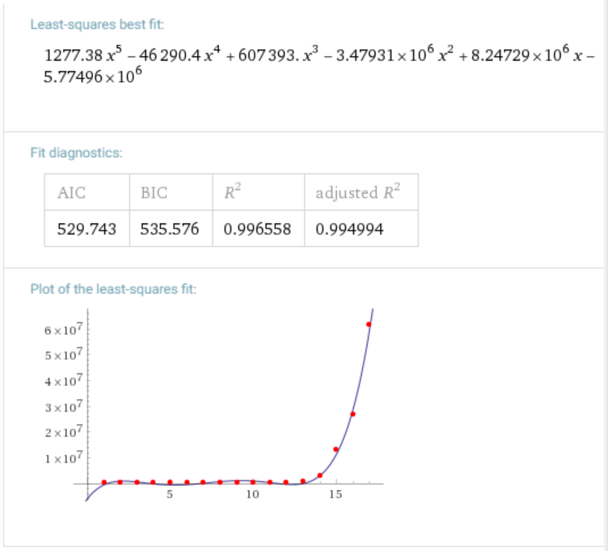
The following images show that the time complexity to solve an instance increases with the number of robots required to solve an instance. The first image shows that for $R = 2$, the solving time is roughly equal toN^2, the next two images show that for $R = 5$, the



$2136.96\,x^2 - 13\,748.3\,x + 126\,688.$

AIC	BIC	R^2	adjusted R^2
270.586	272.846	0.992363	0.990835

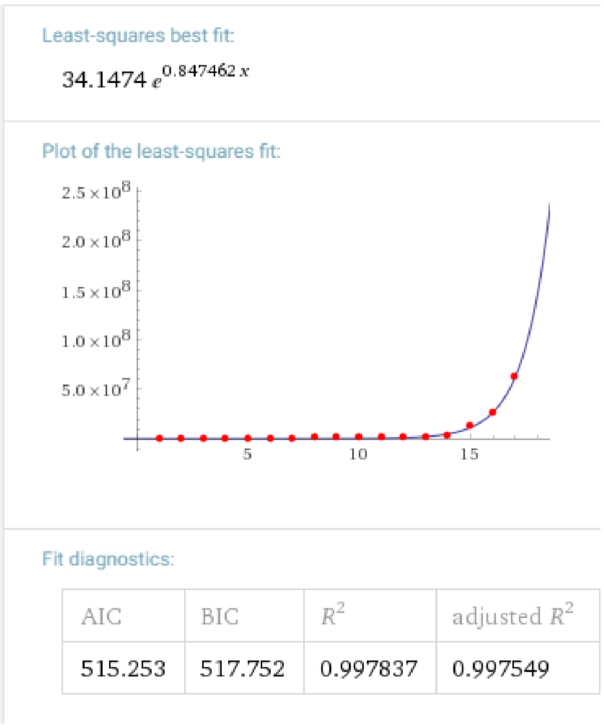
Figure 1: $R = 2$, N^2 Time



$1277.38\,x^5 - 46\,290.4\,x^4 + 607\,393.\,x^3 - 3.47931\times10^6\,x^2 + 8.24729\times10^6\,x - 5.77496\times10^6$

AIC	BIC	R^2	adjusted R^2
529.743	535.576	0.996558	0.994994

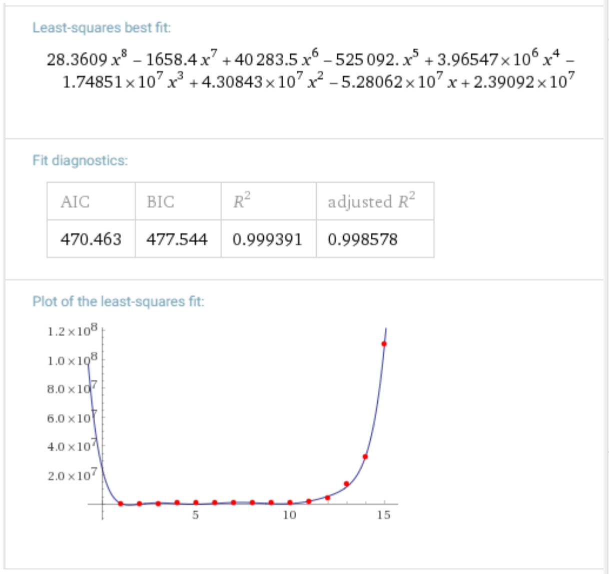
Figure 2: $R = 5$, N^5 Time



$34.1474\,e^{0.847462\,x}$

AIC	BIC	R^2	adjusted R^2
515.253	517.752	0.997837	0.997549

Figure 3: $R = 5$, Exp. Time


Least-squares best fit:

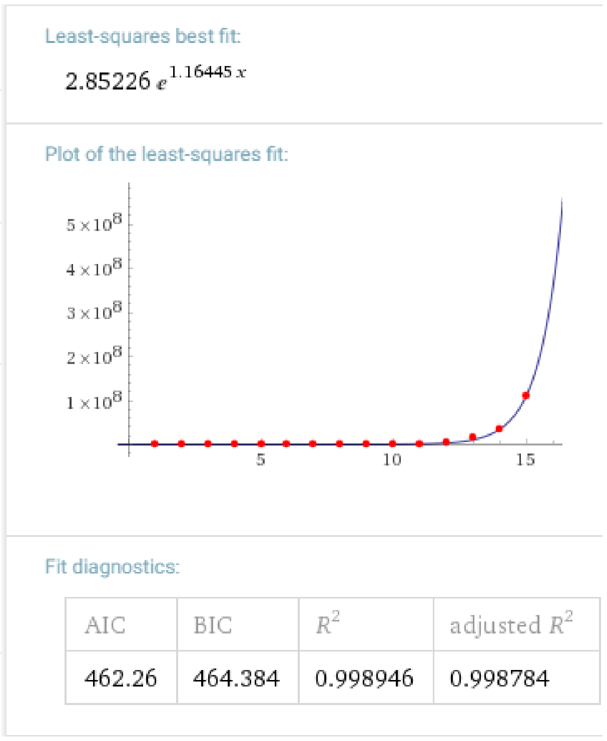
$28.3609\,x^8 - 1658.4\,x^7 + 40\,283.5\,x^6 - 525\,092.\,x^5 + 3.96547\times10^6\,x^4 - 1.74851\times10^7\,x^3 + 4.30843\times10^7\,x^2 - 5.28062\times10^7\,x + 2.39092\times10^7$

Fit diagnostics:

AIC	BIC	R^2	adjusted R^2
470.463	477.544	0.999391	0.998578

Plot of the least-squares fit:

Figure 4: $R = 8$, N^8 Time


Least-squares best fit:

$2.85226\,e^{1.16445\,x}$

Plot of the least-squares fit:

Fit diagnostics:

AIC	BIC	R^2	adjusted R^2
462.26	464.384	0.998946	0.998784

Figure 5: $R = 8$, Exp. Time

solving time is roughly equal to N^5 or exponential time, the last two images show that for $R = 8$, the solving time is roughly equal to N^8 or exponential time.

The above graphs measure steps versus the time that step took to solve in microseconds. Each of the above instances had the same layout, number of goals per robot and warehouse size. I set the instances to time out after about 5 minutes. R=2 finished solving, however R=5 and R=8 did not. These graphs were created in WolframAlpha.

Solving the Time Complexity Issue

Solving the time complexity issue is important because the paper I base this project on uses answer set programming and asprilo. Using another program or technology stack may result in a better solution, but defeats the purpose of this assignment.

My first solution to getting around the time complexity issue was to try better warehouse design. Some examples included placing the station in the center of the warehouse, placing the robots' starting position closer to the shelves and re-arranging the shelves to allow for the algorithm to solve instances with fewer steps. While these changes improved the algorithm's solving time, they did not solve the overall problem and cannot be considered a general solution. For example, an instance with a large number of orders or robots will always create a strain on time complexity.

My main solution to getting around the time complexity issue involves decentralization. Decentralization is used to break up a warehouse into individual cells. Every cell has its own asprilo instance, and robots cannot leave their cell of origin. Decentralization works because we can limit the number of robots used in a particular area, thereby setting an upper limit on time complexity while keeping the amount of work done in the entire warehouse the same. Decentralization also limits the physical size of each cell which limits the maximum number of steps required to move a shelf to a station. Note that due to the stand-alone nature of asprilo instances, each cell need not be identical or even contiguous.

The next factor to consider is order lines. If the number of robots per cell is greater than 1, the number of steps required to solve an instance will eventually cause an additional strain on time complexity. To solve this we are going to limit the number of orders in a cell instance to one per robot. Once these orders are fulfilled, the cell instance will stop computing and restart with the next goal for each robot, this process will continue until all orders are fulfilled. This method works because the overhead cost of recalculating after each goal is preferable to the potentially long solving time required for longer goals.

Another factor to consider is passing shelves between cells. In real world scenarios, products need to be moved from one section of a warehouse to another, however, in our solution, robots may not leave their cell of origin. To solve this, each cell may have an implicit border area with a station. This border area is shared by adjacent cells in order to facilitate moving shelves from one cell to another. When a robot ‘A’ from cell ‘A’ receives an order to pass a shelf to an adjacent cell ‘B’, the ‘A’ simply places the shelf in the proper station in the border area between cells ‘A’ and ‘B’. Once all orders are fulfilled, the shelf is removed from cell ‘A’ and placed in cell ’B’ via a python script. A robot ‘B’ from the cell ‘B’ may receive an order to move the shelf to another station.

Finally, one last factor to consider is passing shelves long distances. The method described previously works well, but may cause a backlog of orders if passing shelves between cells overwhelms orders when shelves are moved inside cells. To solve this, we can add one more instance for the entire warehouse, except this time one node refers to one cell. The robots in this scaled-down instance travel in each border area and are responsible for moving shelves from one cell to another if the required travel distance is long. This method works because there is no given scale between a node in asprilo and a square foot in the real world. Additionally, we can design each cell so that the border areas are empty outside of quick deliveries, so we can assume that the rapid transit robots can pass through cells without issue. If the scaled-down version is too large and slowdowns occur, this method may be used again (an instance showing cells made of cells) until a satisfactory solution is found.

In summary, the combination of efficient warehouse design, decentralization, border areas and scaled-down cells efficiently solve large instances that would otherwise take an indeterminate amount of time to solve.

References

M. Gebser, P. Obermeier, T. Otto, T. Schaub, O. Sabuncu, V. Nguyen, and T. Son, “Experimenting with robotic intra-logistics domains” Theory and Practice of Logic Programming, pp. 502-519, July 2018.