Smart Food Delivery Service

Christian Impollonia, Dana Alsuwailem, Hanna Hiridjee, Sawsan Malak, Nour Rabih

Computer Science BSc – King’s College London  
[christian.impollonia@kcl.ac.uk](mailto:christian.impollonia@kcl.ac.uk), [dana.alsuwailem@kcl.ac.uk](mailto:dana.alsuwailem@kcl.ac.uk), [hanna.hiridjee@kcl.ac.uk](mailto:hanna.hiridjee@kcl.ac.uk), [sawsan.malak@kcl.ac.uk](mailto:sawsan.malak@kcl.ac.uk), [nour.rabih@kcl.ac.uk](mailto:nour.rabih@kcl.ac.uk)

Abstract

This is a report for the INT (Introduction to Artificial Intelligence) coursework at King’s College London. We will go into detail about our planning domain and how the planner handles the various problem files we gave it to solve.

Introduction

Our domain proposition is a food delivery service. Our service allows drivers to pick up multiple orders from restaurants depending on their capacity. With that, drivers are then instructed by the planner to deliver orders and move between clients and restaurants. At the end, each order must be delivered. Delivery services are a continuously growing market, especially with the Covid-19 pandemic. Hence, we believe that our choice of domain is interesting because of the growing need for a reliable, fast and efficient system guiding the delivery process.

Domain choice discussion

We chose this domain because we believe it can have very useful applications in real life scenarios. Services like Deliveroo and Uber Eats are becoming more and more popular every day. Moreover, with the rise of e-commerce, delivery services are booming, and with that being known, the need for a competitive delivery planner seems to be logical. We believe planning is best suited for this kind of system because it creates a list of actions to follow, i.e., a plan that each driver can go by when they are on the job. Our initial idea was for the planner to find the best course of action by minimizing the number of steps the driver could take (as in km driven), but we adapted our domain when we understood that OPTIC could not optimize, so we decided to go for a more simplified version that could later be expanded. Our domain can be used for any delivery service, it is not restricted to being a food delivery service, which makes it more useful and desirable at this time, as companies of all industries, whether it’s fashion, home appliances, flowers, small business, or a big business, are Shifting to online platforms.

Domain richness

Our domain consists of 5 main types: driver, order, client, restaurant, and location (of the client). The drivers, who all start at a restaurant, are the protagonists of each plan, as they are the ones that execute all the actions. Clients place orders that are then picked-up from the restaurant and dropped off to their location by the driver.

Our domain file uses a range of predicates in order to check on the status of our objects at any point in time.

For instance, ‘order\_at\_restaurant’ checks whether the order is at the given restaurant, thus allowing a driver to pick it up. We similarly have another predicate, ‘client\_at\_location’.

The ‘order\_belongs\_to’ predicate is used to link the order to the corresponding client and delivery location, this avoids delivery mishaps (e.g., delivering to the wrong client) and allows the planner to know what location the driver needs to head towards. Another interesting predicate is “picked\_up”. This serves the purpose of denoting that an order belonging to a client is being handled by a driver.

The main actions we have in our domain are “move”, “pick-up” and “drop”. “pick-up” and “drop” are self-explanatory, the first means that the driver will pick up a specific order from a restaurant, and therefore is exclusively used in restaurants, and this is done only under certain pre-conditions which include: the driver being at the restaurant, the order being at the restaurant, and the driver having capacity to carry the order. While the second– drop - means the driver will drop a specific order to a client, so it can only be used when the driver is at the right location, order belongs to the client that is at the location, and that the order has been originally picked up.

To enable the driver to move between the restaurants and locations, we sub-divided the ‘move’ action into 3 in order to more accurately satisfy specific preconditions and effects. The *‘move\_to\_restaurant’* action captures the behavior of a driver moving from a location (where he last dropped an order) to the restaurant, usually to pick up a new order. The *‘move\_client\_to\_client’* action is used when a driver has dropped an order and has another order to drop to a different location. Additionally, we have three actions depicting the driver moving from a restaurant to a client , *‘move\_restaurant\_client\_maxCap’, ‘move\_rest\_client\_NoOrders’, and ‘move\_restaurant\_client\_maxCapNoOrders’.* The driver can move from the restaurant to the client’s location if his maximum capacity has been reached (using *‘move\_restaurant\_client\_maxCap’)*, if the restaurant has no more orders to pick up and he doesn’t need to visit another restaurant (using *move\_rest\_client\_NoOrders’),* or if both cases hold (*‘move\_restaurant\_client\_maxCapNoOrders*). The need for the three versions of this action stemmed from the issue we were facing with the driver only picking up and dropping one order at a time, as well as not traveling to other restaurants to pick up orders. Finally, we have ‘*move\_rest\_rest’*, which is used for the driver to travel from one restaurant to another to pick-up more orders. With these additions, the planner more accurately simulates a real-life scenario.

Used PDDL features

We decided to use **typing** in our domain to be able to have different objects with different characteristics, as this was the best way to represent drivers, orders, clients, locations, and restaurants. We thought of using inheritance to represent drivers and clients as persons, and locations and restaurants as places, but we did not see how we could take advantage of this in any way, so we preferred having 5 different types. To help with the fact that OPTIC does not support predicate negation, we decided to use **fluents**, and have 1 represent true and 0 represent false. We tried the option of using two predicates: one being the regular one and the other representing negation, but we could not find a way to make them mutually exclusive: i.e., sometimes they were both given a value of true at the same time, and we could not negate them. So, by using functions like ‘orderDelivered’, 1 represents that the order is delivered to the client and 0 represents otherwise. Likewise, by utilizing ‘*isAtLoc*’ we could increase it by 1 when the driver arrived at a location and decrease it when they arrived at a restaurant, eliminating the problem of a driver being in two places at once. Because of this approach, another issue arose when moving from client to client. We had to decrease ‘*isAtLoc*’ of the departing location by 1 and increase ‘*isAtLoc*’ of the arriving location by 1 despite passing different variables (“to” and “from”). The planner read this as changing the value of a function twice simultaneously and could not do it. So, we decided to implement **durative actions** to solve this. The action move client to client has a durative effect: it decreases ‘*isAtLoc*’ of “from” by 1 at start, and then increases ‘*isAtLoc*’ of “to” by 1 at end. Similarly, when moving from one restaurant to another, a durative action was also used to prevent the same issue of not being able to decrease ‘*isAtRest*’ of the departing restaurant by 1 and increase ‘*isAtRest*’ of the arriving restaurant by 1. Therefore, the durative action decreases ‘*isAtRest*’ of “from” by 1 at start, and then increase ‘*isAtRest*’ of “to” by 1 at end. This made it work as we intended. Fluents were also used to keep count of the ‘driverCapacity’ and ‘capacityUsed’, these two functions help us make sure that the driver is not overloaded with orders. Two more fluents/functions are used, which are ‘numOrdersRest’, which represents the number of orders at a restaurant, as well as ‘numberOfRest’, which is the number of restaurants that still have unfulfilled orders that need to be picked up. The overall use of functions has been the key to the functionality of our domain.

Part 2:

Efficacy of problem files

We decided to create a range of problem files to represent the various ways our planner could work in a real-life scenario. Meaning, by gradually increasing the number of objects in the world. So, we have extremely simple situations with 1 driver, 1 restaurant and 1 client with 1 order, which shows the basic functionality of the program. Then we test what would happen with 2 or more orders, depending on driver capacity as well, meaning picking everything up at once, or having to come back. We have 2 or more drivers, that would split the orders between them, or with 2 or more restaurants, so not going to the same place every time to pick up food. We also test situations where the same client makes two orders, so the driver needs to drop twice at the same location. We believe testing for these possible edge cases makes sure that we have a solid domain, as they all represent slightly different executions of the planner, instead of just increasing the numbers proportionally.

Table

Description automatically generated

Solvable problems

Our domain can solve most of the problem files we thought of. The most basic one, with 1 of every object, is obviously easily solved. And this stays the same even when we increase the number of clients and orders. If the capacity of the driver allows it, they will pick up as many orders as they need and then go to each location to drop it. In case the driver’s capacity is not enough (for example with 3 orders, 3 clients, 1 restaurant and 1 driver with capacity of 2) the driver will pick all the orders they can carry, deliver them, and then go back to the restaurant to pick up the remaining ones. In case there is more than one restaurant, the driver will pick up what they need from one and then go to the other restaurant if there are any orders there. In case a client makes two orders, the driver will drop them both at the client’s location with no issues. It can also happen that two clients share the same location and make two orders; in that case the driver will reach the location and deliver each order to the respective client. We’ve also tested what happens when we add irrelevant objects to our problem, like a restaurant with no orders. If that happens, they will just be ignored by the driver. We’ve gradually increased the number of orders to handle, and also tried with larger driver capacities, and they can all be solved.

Valid but insolvable problems

One problem that we found to be unsolvable is when a driver starts at a location that is not a restaurant. When a driver starts at a client’s location, it cannot go back to the restaurant as it does not meet all the preconditions of “move\_to\_restaurant”. Therefore, to avoid this problem we’re assuming that a driver should always be at a restaurant at the start.

All the other problems we created to test our domain passed, so we did not find any other insolvable problems. We did however encounter some problems where the solution found worked but wasn’t the optimal one. Our domain had some issues handling multiple drivers delivering at the same time. Mainly, the final goal is achieved (all orders are delivered and both drivers are working) but it is not done simultaneously, which is not ideal. We believe a way to solve this would be by having a time variable, and to have the planner optimize of it, by trying to find the solution that requires the least amount of time. As we understood though, this is not possible for this version of OPTIC, so our planner remains a guide for a single driver’s journey and should be used individually. We also think we would need something like threads, for a planner to handle different agents acting simultaneously, and we were not sure about how to implement that.

Final analysis on planner

Our planner generally solves all problems we gave it very quickly, as it is not exploring a huge number of possible solutions, but it is heavily constrained to do what we ask of it. It is optimal to a certain extent, where it picks up the orders from one location in one go, rather than keep going back and forth. If it were allowed to optimize, we believe we could have optimal solutions that we don’t have for cases like the multiple drivers. We could also simplify the code in our domain quite a bit, as we would not have to force our planner to, for example, pick up as many times as it can. It would do it automatically to minimize steps and time taken. We also have not managed to find a limit for the number of objects the planner can handle, as it even solves a problem with 40 orders, 40 clients, 40 locations and 20 restaurants in seconds. Another feature that can be added, would be considering the order processing time. Our current domain file assumes that all orders are ready, however having a different preparation time for each order and the time the order was placed, would make our domain more efficient and useful. In conclusion, we believe our domain is a fairly simple one, but easily extendible with more powerful features, and also very versatile, as it could be modified to act as any delivery service, with useful applications in the real world.