GENERAL INFO

Home

Members

News

Contact

ACAI RL Summer School

RESEARCH

Publications

Topics

Projects

Software

FOR STUDENTS

Courses

Thesis proposals 2017-2018

LINKS

Evolutionary Linguistics
Fluid Construction Grammar
Robotics wiki
IB2 - Bioinformatics Institute
WWCS 2015
DynaMine

Declarative Programming Project: Delivery Planning

The project presentation can be downloaded here: project presentation

For the project you'll have to implement a delivery planning tool using SWI-Prolog.

Introduction

The Delivery Planning Problem

Assume you own a company with a set of depots storing goods, and a fleet of vehicles that can be used to transport them. You've received a set of orders through your sales channel that must be delivered. When should which orders be shipped? From what depot? Using which vehicle? And in what sequence should they be delivered?
Making a good plan is challenging due to the often large amounts of orders that must be delivered before a certain deadline, as to avoid late fees. Depots have a restricted inventory and your fleet consists out of a finite number of vehicles, each having a limited speed and capacity. In addition, their use incurs costs (gasoline, driver's fee), which gives rise to a tradeoff between delivering goods on time (if at all) and the cost of doing so. Delivery Planning represents a challenge for many companies, and (partial) automation of this task is an active research area.

Problem Instances

To test your planner, a set of problem instances are provided.

Each specifies the following knowledge:

- A set of product types:
product(PID,Value,Weight): A product type with unique identifier 'PID', of which each instance is sold for 'Value' euro and weighs 'Weight' kg.

e.g.

```
product(p1,10,0.1). %items of product p1 weighs 100gr and cost €10.
product(p2,80,25). %items of product p2 weighs 25kg and cost €80.
```

- A set of orders:
order(OID,OrderDetails,Location,Deadline): An order with unique identifier 'OID', where 'OrderDetails' is of the form [PID1/Amount1,...,PIDn/Amountn]
and indicates how many items ('Amounti') of each product 'PIDi' are ordered. If some product PID does not appear in this list, no items of product PID are ordered, i.e. PID/0.
The order is to be delivered at 'Location', and this at day 'deadline' latest.

e.g.

```
order(o1,[p1/2],location(2,2),1). %order o1 consists of 2 items of p1
and must be delivered before day 2 of the planning period.
order(o2,[p2/1],location(2,3),2). %order o1 consists of 1 item of p2.
and must be delivered before day 3 of the planning period.
```

- A set of depots:
depot(DID,Inventory,Location): A depot with unique identifier 'DID'

located at 'Location', where 'Inventory' is (~OrderDetails)
of the form [PID1/Amount1,...,PIDn/Amountn] and specifies how
many items ('Amounti') of each product 'PIDi' are available at
the depot, at the beginning of the planning period.

e.g.
```
depot(d1,[p1/7],location(4,3)). %depot d1 has 7 items of p1.
depot(d2,[p2/5,p1/1],location(1,1)). %depot d2 has 5 items of p2
and 1 of p1.
```

- A set of vehicles:
vehicle(VID,StartDID,Capacity,Pace,UsageCost,KmCost):
A vehicle with unique identifier 'VID', located at depot 'StartDID'
at the beginning of the planning period.
It can carry up to 'Capacity' kg and moves at 'Pace' min/km.
'UsageCost' euro must be paid each day it is used and an additional
'KmCost' must be paid per km driven.
e.g.
```
vehicle(v1,d1,100,0.75,50,0.5). %vehicle v1, is at the beginning of
the planning period located at depot d1. It has a capacity of 100kg
and moves at a pace of 0.75 min/km (80 km/h).
It costs €50 for each day of use, and €0.50 per km driven.

vehicle(v2,d2,200,1,50,0.5). %vehicle v2 differs from v1 in that it
is initially located at depot d2, has a capacity of 200 kg and moves
at 1.0 min/km (60 km/h).
```

- A set of working days:
working_day(Day,StartTime,EndTime):
Indicates that all transport on day 'Day' of the planning period
must take place in the [StartTime,EndTime] window, where time is
indicated in minutes.
If for some day no working_day/3 fact
is specified,
no transport can take place that day.

e.g.
```
working_day(1,540,1020). %on day 1 vehicles can leave the depot as of
of 9am and must return to a depot by 5pm latest.
working_day(3,360,1080). %on day 3 vehicles can leave the depot as of
6am and must return to a depot by 6pm latest.
%no transport can take place on day 2 (or 4,5,...).
```

The following table gives an overview of the instances provided:

#	name	# orders	# product types	# depots	# vehicles	# working days (hours)	optimal profit
1	single_small	10	3	1	1	1 (12)	€1392.8
2	multi_depots_small	10	3	3	1	1 (10)	€878.0
3	multi_vehicles_small	10	3	1	3	1 (6)	€1128.2
4	multi_days_small	10	3	1	1	3 (18)	€900.4
5	multi_small	10	3	2	2	2 (12)	€804.2
6	single_large	100	4	1	1	1 (12)	???
7	multi_depots_large	100	4	5	1	1 (12)	???

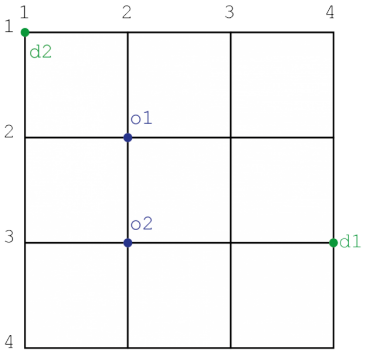
8	multi_vehicles_large	100	4	1	5	1 (10)	???
9	multi_days_large	100	4	1	1	5 (50)	???
10	multi_large	100	4	5	5	5 (50)	???

Download all as a zip archive here.

In these instances the road network is modeled as a 2D grid, where each cell edge has a length of 1 km. Delivery destinations and depots are located at their intersections, indicated using the `location(X,Y)` functor, specifying a location at the intersection of the X^{th} vertical street and y^{th} horizontal street. Vehicles are assumed to move at a constant pace (see vehicle/6 predicate) and follow the shortest path. The driving distance between 2 locations is therefore equal to the manhattan distance.

E.g.

Given the example facts above, the corresponding road network looks as follows:



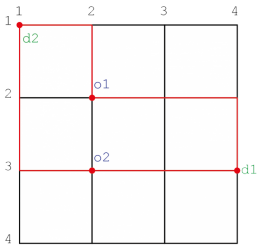
The driving distance from depot d1 to depot d2 is 5 km, and will take us 3 minutes and 45 seconds using v1.

Solution Format

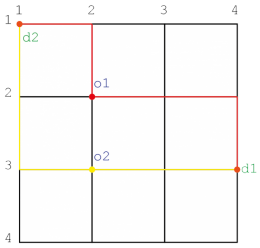
A delivery schedule must be represented as a `plan/1` functor, containing a list of `schedule/3` functors, one for each vehicle, for each working day. A `schedule(VID,Day,Route)` functor indicates that vehicle 'VID' follows delivery route '`Route`' at working day 'Day'. Here a route corresponds to a list of order and/or depot IDs that must be delivered and/or visited in order.

e.g.

```
%example schedule A
plan(
    schedule(v1,1,[o1,d2,o2,d1]),
    schedule(v2,1,[]),
    schedule(v1,3,[]),
    schedule(v2,3,[])
)
```




```
%Example schedule B
plan(
    schedule(v1,1,[o1,d2]),
    schedule(v2,1,[]),
    schedule(v1,3,[]),
    schedule(v2,3,[o2,d1])
)
```



Hard-constraints

The following constraints must be met in order for a plan to be admissable:

1) It must have exactly one schedule for each vehicle on each working day. Note that if a vehicle is not used some working day, a schedule with an empty route must be provided (as in the examples above).

2) Each order is delivered at most once (as a whole, not item per item). This means that orders may be delivered late or not at all.

3) Transport can only occur during the working day:

- A vehicle cannot leave the depot it parked last working day (or load any items) before the 'StartTime' (see working_day\3 predicate). See the vehicle/6 predicate for its location at the beginning of the planning period.
- A vehicle must arrive back in a depot at 'EndTime' latest. Note that this can be a different depot from which it left in the morning.

This means that the total duration of a vehicle's route must not exceed EndTime-StartTime. Both driving and loading/delivering orders takes time. The duration of a route is

`total_distance_driven x pace + 10 x #orders_handled`

minutes, where vehicle moves at `pace` min/km and loading and delivering an order take 5 minutes each.

E.g. In example schedule A, the route of v1 at day 1 takes
`10km x 0.75 min/km + 10 min x 2 = 27.5 min`

4) Vehicles are only allowed to visit a depot empty (end of day/or reloading) and do not carry goods from depot to depot. This means that delivered goods are always taken from the inventory of the last visited depot.

E.g. In example schedule A, o2 is loaded in v1 at d2, not at d1.

5) An item can only be taken from a depot's inventory if it is (still) available. Depots are not restocked during the planning period.

E.g. the following schedule is invalid:

```
plan(
    schedule(v1,1,[o1,o2,d2]),
    schedule(v2,1,[]),
    schedule(v1,3,[]),
    schedule(v2,3,[])
)
```

as o1 contains 2 items of p1, while only 1 is available at d1.

6) A vehicle can only transport orders, if the total weight of the items therein does not exceed its capacity (see vehicle/6 predicate). This means that the load of a vehicle is the total weight of orders still to be delivered before visiting a depot (end of day/or reloading).

E.g. assume an a third order

```
order(o3,[p2/4],location(3,3),2).
```

the following schedule is invalid

```
plan(
    schedule(v1,1,[o1,d2,o2,o3,d1]),
    schedule(v2,1,[]),
    schedule(v1,3,[]),
    schedule(v2,3,[])
)
```

because after visiting d2, v1 would have o2 and o3 on board, weighing 125 kg in total and thus exceeding v1's 100 kg capacity.

On the other hand, the following schedules are valid

```
%first load and deliver o2, before loading o3
plan(
    schedule(v1,1,[o1,d2,o2,d2,o3,d1]),
    schedule(v2,1,[]),
    schedule(v1,3,[]),
    schedule(v2,3,[])
)
%deliver o2 and o3 using v2 (capacity of 200 kg)
plan(
    schedule(v1,1,[]),
    schedule(v2,1,[o2,o3,d1,o1,d2]),
    schedule(v1,3,[]),
    schedule(v2,3,[])
)
```

Optimization objective

Often many different admissible plans exist, which are not all as good. E.g. a plan which fails to deliver any order satisfies all hard constraints... In particular, we're interested in finding a plan that maximizes our company's ***`Profit = Revenue - Expenses`***

Revenue is made by delivering orders timely. Concretely, if we deliver an item before (or on) its deadline we receive the full value of the order (sum of values of all items therein). If we deliver an order late (i.e. after the deadline), we receive only half its value. If we fail to deliver an order we earn nothing.

Expenses are made to transport the items. Using a vehicle for a working day incurs a fixed cost (e.g. fixed driver fee) as well as a cost for per km driven (e.g. gasoline, hourly driver fee). Concretely, the cost of using a vehicle on a working day is equal to `UsageCost + total distance driven x KmCost`. If for some day, a vehicle is not used (`Route = []`), no costs are incurred.

E.g.

For example schedule A:

```
revenue: €100
- o1 on time: €20
- o2 on time: €80
cost: €55
- using v1 for 1 day (€50)
- driving 10km using v1 (€5)
profit: €45
```

For example schedule B:

```
revenue: €60
- o1 on time: €20
- o2 late: €40
cost: €105
- using v1 for 1 day (€50)
- using v2 for 1 day (€50)
- driving 5km using v1 (€2.5)
- driving 5km using v2 (€2.5)
profit: €-45
```

Functional Requirements

Write a Prolog program to solve the Delivery Planning Problem.

Concretely, you must implement the following predicates:

Auxiliary Functionality {10%}

These are relatively easy to implement and should get you started.

`driving_duration(+VID,+FromID,+ToID,-Duration)`: Where Duration is the (shortest) time it takes vehicle VID to drive from depot/order FromID to depot/order ToID. {2.5%}

`earning(+OID,+Day,-Value)`: Where Value is the revenue received for delivering order OID on day Day. If Day is not a working day it should fail. {2.5%}

`load(+Os,-Weight)`: Where Weight is the total weight in kilograms of the items in the list of orders Os. {2.5%}

`update_inventory(+Inventory,?OID,?NewInventory)`: Where NewInventory is the inventory after taking order OID from Inventory. If insufficient items are available it should fail. {2.5%}

Core Functionality {80%}

These represent the core functionality your tool should offer and account for the large majority of your grade.

`is_valid(+P)`: Checks whether plan P is valid, i.e. is of the correct format and satisfies all hard constraint. Note that schedules may appear in any order (i.e. all permutations are valid). {15%}

`profit(+P,-Profit)`: Where Profit is the profit of valid plan P. {15%}

`find_optimal(-P)`: Finds a plan P, exactly maximizing profit. {20%}

`find_heuristically(-P)`: Finds a valid plan P, approximately maximizing profit. Note that while the quality of the plan found by find_heuristically/1 will be taken into account while grading, you can already earn points by returning an arbitrary valid plan. {20%}

`pretty_print(+P)`: Outputs a valid plan P nicely, in a human readable format. The output should, for each working day, provide detailed information about the route each vehicle used that day, has to follow, indicating where and when, which orders should picked up/delivered (example see presentation slides). {10%}

Extended Functionality {up to 15%}

Implementing extensions can earn you up to 3 additional points. Below are some suggestions (for 2 points), but we invite you to be original and come up with your own. However, make sure to retain the base functionality.

`is_valid(?P)`: is_valid/1 next to checking, also generates all valid plans (without literal duplicates) in under 3 minutes for small. To achieve this, you're allowed to only generate semantically different plans (e.g. schedules in a fixed order). Make sure it still accepts any valid plan (e.g. any permutation of schedules). {5%}

`is_optimal(?P)`: Checks whether plan P is optimal. If P is an unbound variable, is_optimal/1 generates all optimal plans exactly once (in under 3 minutes for small). {5%}

...

To help you veryfing the correctness of your implementation some examples of driving distances, weights, earnings, (in)valid plans and their profit can be found here.

Non-Functional Requirements

It also works for us...

The program has to be written in SWI-prolog and should run on the computers of the rooms of IG. Make sure that your source files are standard Unix text files, e.g. lines are separated by newline characters only. For those of you that make their project at home and/or with a different prolog, make sure that your code is fully functional on the target system. Your code must work when run using SWI-prolog in the computer rooms! Excuses of the “it worked at home but not on this system” type will not be accepted. ***If it doesn't work for us, it doesn't work*.**

Generality

The same code should be able to solve all given problem instances. Furthermore, to verify generality, your code will be tested on some additional instances. It therefore shouldn't rely on (or exploit) specific properties of these instances.

Procedural style

By using cuts, assert/retract and the build-in if-statement it is possible to write Java-like programs. This style makes hardly any use of the declarative features of prolog, and such programs are better written in procedural languages. As this is a declarative programming course, the latter is strongly not recommended.

Efficiency

All predicates must run in under 2 minutes on the lab computers, for all instances (with exception of find_optimal/1 for the large instances, see final section).

You are thus to find a careful balance between declarative style, readability & efficiency:

E.g. Use red cuts if and only if they have a strong impact on efficiency.

Modularity

Work modular:

- Define your solution in a different file than the instances
- Divide largely independent logic for your solver over multiple files (e.g. 1 for each requested predicate).
- Use prolog modules instead of consult/include to combine the seperate parts of your solution.

Code Duplication

Closely related to modularity, code duplication is to be avoided as well. Having small reusable predicates will not only improve the quality of your code, it will also make things easier for you and help you avoid typical copy-paste related errors.

Syntax Conformity

Make sure you use the same syntax as used in the assignment:

- For requested predicates (e.g. `is_valid` and not `isValid`)
- For plans (e.g. `plan([schedule(v1,1,[])])` and not `[schedule(v1,1,[])].`

Internally you're encouraged to use your own predicates and data structures (whatever is most convenient/efficient), but make sure you at least provide the requested predicates, using the requested syntax. This primirarly because we'll use unit-tests to verify the correctness of your implementation.

Commenting

Comment your source code. Writing good comments is an art, even more so in Prolog. Try therefore to follow the prolog-specific commenting guidelines given here as much as possible.

Reporting Requirements

Next to your implementation, you are to write a (brief) report containing:

- A brief description of your solution approach (design, data-structures, algorithms)
- Clearly state the strengths and weaknesses of your implementation. E.g.
  - Which predicates work? (For which instances?)
  - Have you implemented any extensions?
  - Quality of heuristic solutions to large instances?
  - Non-functional requirements?
- Experimental results:
  - For the small instances: An optimal solution found by your scheduler *
  - For the large instances: The cost of your heuristic solution *

* Note that any reported experimental results must be reproducable in under 3 minutes on the lab computers AND at the oral defense.

Deadline for 2nd Session

The firm deadline for this project, in the second session, is **Thursday, 17 August 2017**, at midnight. The defenses will be held the 23th of August 2017.

Remark: As a different professor will be teaching this course next academic year (2017-2018), we cannot assure transfer of partial grades for written exam/project (e.g. if you fail the written exam, pass the project and fail the course overall, you may still need to redo the project next year)

You should submit a zip archive containing the following deliverables on Pointcarré:

- Your source code files: well structured and with the necessary documentation.
- A report in PDF that briefly describes the design and functionality of your program.
- A small manual and an example run of your program: the idea is that I should be able to test your program by myself, without having to contact you for additional information.
- Everything must be submitted using Pointcarré (English course page, tools tab under Assignment) any time before the deadline**.**

If you have any questions related to the project assignment or Prolog in general, feel free to contact me by email (steven.adriaensen@vub.ac.be)

Some important notes

Grading

Functionality accounts for a large part of your final grade. Make sure to carefully read the assignment and verify whether everything works according to specification. To aid you in this, we provide you with some correct input/output examples here. Also, the optimal profit for all small instances are given in the table above. In your report, clearly state which of the requested predicates (don't) work. If some work only partially (e.g. on only a subset of instances) , state exactly when they do(n't) and make it easy to test this. Not all requested functionality weighs as strongly and their relative contributions are specified in the Functionality section. As extended functionality you are free to implement any predicates you deem fit, which (depending on their originality, difficulty and relevance) can earn you up to 3 additional points (to a total of at most 20/20...).

While functionality is important, non-functional requirements and the quality of your deliverables in general (report, manual, source code) will also be taken into account when grading.

Submitting Late

Late submissions are not allowed and might be rejected. If accepted, late penalties will always apply. It is your own responsibility to confirm that your submission was succesful and complete.

Optimal Solutions for Large Instances

The optimal solution must only be reported for the small instances (as solving the large instances exactly under 3 min is not feasible in general). For the large instances only the profit of the plan found by your heuristic must be reported.

Cuts

Use cuts in a correct way and at the correct place. When a rule is not supposed to backtrack put the cut inside that rule, and not when you use the rule:

```
foo(X) :- dont_backtrack(X), !
```

vs:

```
foo(X) :- dont_backtrack(X).  bar(X) :- foo(X), !.
```

Retract after/before assert

Sometimes a predicate can be implemented more easily/efficiently using assert. While useful when done properly, it has various risks:

- It potentially creates dependencies between queries. Make sure the requested predicates work individually (without requiring other predicates to be ran first)
- While correcting your assignment, we'll be dynamically loading/unloading different instances. Instance-specific memoized data can cause havoc in this setting.

One way to avoid these problems is to make sure that each of the requested predicates cleans up after itself, i.e. retracts all asserted data. Another approach is to retract all dynamically asserted data which might interfer with a predicate, beforehand.