assignment 5 - prolog

fredrik robertsen

2025-11-05

task 1

the following code is my solution:

```
payment(Sum, Coins) :-
    payment_acc(0, Sum, Coins).

payment_acc(Acc, Sum, []) :-
    Acc #= Sum.

payment_acc(Acc, Sum, [coin(Count, Value, Available)|
Tail]) :-
    Count in 0..Available,
    NewAcc #= Acc + Count * Value,
    payment_acc(NewAcc, Sum, Tail).
```

the predicate `payment/2` has two arguments (hence `/2`). the first, `Sum`, is a target cost we are attempting to make from a collection of `Coins`, a list of `coin/3` items, specifying the value of the coin type and how many of such coins we have available.

this is a classic constraint satisfaction problem which is solved elegantly in prolog, even with my messy first-timer code above.

we are essentially making a search in the state graph of our problem space. this is what prolog does when it performs a combination of infer-

ence/deduction, DFS and backtracking to find values that fit our variables, given the stated predicates.

in this program we create a helper function which `accumulates` a sum accross the recursion. this is similar to how we would solve it in oz, except we would need to implement the search explicitly. prolog does a lot of heavy lifting in that regard, since we are essentially only recursively creating a long string of CNF logic statements that constrain our problem until a solution can be found.

note that i am also using patternmatching to destructure the arguments of the `payment_acc` predicate.

task 2

subtask 1

this is my solution code:

```
% arity 4 predicate
plan(Cabin1, Cabin2, Path, TotalDistance) :-
    plan(Cabin1, Cabin2, [Cabin1], Path, TotalDistance).

% base case
plan(Cabin1, Cabin2, Visited, Path, Distance) :-
    not(Cabin1 = Cabin2),
    distance(Cabin1, Cabin2, Distance, 1),
    append(Visited, [Cabin2], Path).

% recursive case
plan(Cabin1, Cabin2, Visited, Path, TotalDistance) :-
    not(Cabin1 = Cabin2),
    distance(Cabin1, CabinX, Distance, 1),
    \+ member(CabinX, Visited),
    append(Visited, [CabinX], NewVisited),
    plan(CabinX, Cabin2, NewVisited, Path, SubDistance),
    TotalDistance is Distance + SubDistance.
```

as you can see, i learned that you can overload predicates with differing arities, such that the previous `payment_acc` could've only been named `payment`. we can also do multiple definitions to clearly state the different branches of a recursive algorithm, such as the base case and the recursive step.

we can then use a `plan/5` auxiliary function to carry a log of what cabins we have already `Visited`. thus, our base case becomes the case where we have a direct connection between the first and last cabin, such that we can easily read the distance from the predicate. then just make sure to mark `Cabin2` as visited.

in the recursive step we assume there is a `CabinX` that lies between our start and end cabins. this cabin cannot have been visited previously, lest we enter an infinite cycle – `\+ member(CabinX, Visited)`. we can then visit `CabinX` and continue our search recursively from there – `plan(CabinX, Cabin2, ...)`. lastly, we can calculate the `TotalDistance` as the sum of the total distance from `CabinX` to `Cabin2` and the total distance from `Cabin1` to `CabinX`.

```
Cabin1  -->  CabinX  -->  Cabin2
   L Distance J L SubDistance J
   L      TotalDistance       J
```

subtask 2

my initial solution was:

```
bestplan(Cabin1, Cabin2, ShortestPath, ShortestDistance) :-
    findall([Path, Distance], (plan(Cabin1, Cabin2, Path,
Distance)), Solutions),
    shortestpath(Solutions, [ShortestPath,
ShortestDistance]).

% takes a list of [Path, Distance] pairs
shortestpath([[Path, Distance]|[]], Solution) :- Solution =
```

```
[Path, Distance].
shortestpath([[Path, Distance]|Tail], [ShortestPath,
ShortestDistance]) :-
    shortestpath(Tail, [TailPath, TailDistance]),
    (Distance < TailDistance ->
        (ShortestPath = Path, ShortestDistance = Distance)
    ;
        (ShortestPath = TailPath, ShortestDistance =
TailDistance)
    ).
```

but i cleaned it up to this:

```
bestplan(Cabin1, Cabin2, ShortestPath, ShortestDistance) :-
    findall([Path, Distance], (plan(Cabin1, Cabin2, Path,
Distance)), Solutions),
    sort(2, @=<, Solutions, [ShortestSolution|_]),
    [ShortestPath, ShortestDistance] = ShortestSolution.
```

the main idea is a bit naive: we are just picking out the shortest path from *all* the paths we find using the `plan` predicate from the last subtask. it sounds like it wouldn't be particularly performant, but i think prolog does a lot of work such that it isn't too terrible.

anyway, i initially just expanded all the paths found by `plan`, then used a homemade `shortestpath/2` predicate that works on a list of `[Path, Distance]` pairs to tail recurse and find the shortest such pair, keeping a running shortest distance result. this is similar to oz, taking the head of the list, checking if it is smaller than the current accumulated value, then carry on the smaller value of the two. the base case is when we only have a single path and distance: we return that as a solution.

but this entire process can be shortened to simply sorting the solutions based on the distance and then picking the first one of that list. the code above performs such a sort and only takes the first solution in the sorted

list, sorting from low to high based on the key 2, such that we are sorting the distance.