Vrije Universiteit Brussel

Project Declarative Programming

Roeland Matthijssens

January 13th

1 Approach of the algorithm

1.1 Data structures

In our program we use two important data structures.

- Taxi
- Customer

Taxis are in the form of

```
1 (Id, Position, Time, Log, Customers_on_board, Back_home)
```

Where Time is the current time for the taxi. since we are going to let all the taxis drive different distances, there is no overarching notion of time. Each taxi will keep track of time himself, and based on his personal time, he can deside which customer is optimal to pick up at that point.
Customers_on_board represents the customers the taxi is currently transporting. This will be a list of customer-Ids. We will use this to determine when the taxi is forced to drop off some customers.
Back_home is used as a flag to tell us whether the taxi is still running around, or if he is already back home. Since we don't send taxis home, unless there is absolutly no work he can do anymore, we can safely ignore taxis that have this flag set.
The position is used to get information about the shortest distance to the customers, and the Id is used for lookup and printing.
The log is used to keep track of what a taxi has done so far. Keeping the log is important, since we are trying to minimize the total waiting time, we need to be able to roll back a taxi to a certain point in his search space when we find another taxi that can perform some action better. Therefor we made it so that each log entry knows the current state, and the previous state.

Customers are in the form of

```
1 (Id, Position, Online_time, Offline_time, Pickup_time, Pickup_taxi).
```

Where Position and Id are trivial.
Online_time and Offline_time represent the times when the customer can be picked up, at the earliest and at the latest.
Pickup_time represents the time when the customer was picked up. And Pickup_taxi is the taxi that is currently picking up the customer.

1.2 The main idea

We approached the problem as a minimization problem. We try to minimize the total sum of time waited by both customers and taxis. In other words we try to use the fullest of our resources (by making the taxis wait as little as possible) and to maximize our efficiency (by making the customers wait as little as possible).
A minimization problem means we search for a possible sollution, and then iterate over this solution to improve it. We do this by iterating over our taxis, and for each taxi we try to find the best customer. We then register this customer as picked up, and continue to iterate over the taxis.

1.3 Customer selection

To get the best customer we first filter the obvious taxis, i.e. the ones for which the current taxi will be to late. We then filter all the customer which we can reach, but someone else is already picking him up, and his arival time is better then the current taxi's would be.

```
best_customer(Taxi_info, Customer_list, Customer_id, Value, Arival_time):-
  get_taxi_info(Taxi_info, _, Taxi_position, Current_time, _, _, _),
  exclude(to_late(Taxi_position, Current_time), Customer_list, Customer_list2),
  exclude(someone_else_is_better(Taxi_position, Current_time)
          , Customer_list2, New_customer_list),
  find_best_customer(Taxi_info, New_customer_list, Customer_id, Value, Arival_time).
```

This gives us a list of potential customers that are elegible for pickup. So we get the 'best' out of this list. We do this by minimizing the wait time of both the customer and the taxi.

```
get_weight(Distance_to_customer, Online_time, Offline_time, Current_time, Result):-
  Arival_time is Current_time + Distance_to_customer,
  Result is ((Arival_time-Online_time)
             /
             ((Offline_time-Online_time)+1))*(1/(Distance_to_customer+1)).
```

This gives us a weight for how good we asume the customer is for the current taxi. If we map this over the remaining customers, we can just take the maximum of this list, and we have the best customer.

1.4 Taxi update

When we have our best customer, either this customer is not yet picked up by any other taxi, in which case we can just claim him, and fill in our arival time for reference by other taxis. In the other case we have a customer that was previously picked up by another taxi. But since we filtered all the customers we couldn't improve we can safely say that we are better than the previous taxi.
Since the customer knows the taxi that was picking him up, we get the other taxi from the list, and we update his position and time, so that it represents the situation right before he picked up the customer.
This is why we need the taxis to keep track of their log.

1.4.1 Roll back log

We know that a taxi can only pickup a customer once. This means we can look through the log for the entry that defines the pickup of the customer. The first part of the log will remain as the new log for the customer, and tail of the log from that point will be rolled back. So we reverse the log, and loop over each entry of the log, and undo each of the actions in reverse order. This leaves the taxi in the state it was before he went to pickup the customer.

1.4.2 End condition

If we repeat this process eventualy we will end up in either of two cases. The first is where the taxi has reached his maximum capacity. The other case is where there are no more customers to pickup for a taxi. In both case we can start to drop off customers. If the taxi has no customers on board, we will let the taxi drive home, and this is also our end condition. When a customer is home, the 'Back_home' flag will be set. The taxis that are back home will be ignored in the next iteration. This does not mean that we will no longer iterate over them. When some other taxi improves a customer this taxi is picking up, we can undo the fact that he is going back home, and in turn unflagging 'Back_home'. From that point on he will again search for the optimal list of actions.

1.4.3 Drop off customer

When we want to drop off a customer, we can just get the best customer that we have on board, and drive to his destination. Since we want to maximize the use of the taxis, the best customer, is just the customer that is closest to the taxi.

2 Dijkstra

We opted for dijkstra over the more directed A* for two reasons. The main reason is that we canŕ really find a heuristic that ensures an underestimate of the distance. Since we have no reliable information of the graph we are dealinig with. A* in real world applications uses the heuristic that the staight distance between two nodes is always smaller or equal to any path between those two nodes. This asumption does not hold for our randomly generated graph. This means that we would have to change the heuristic, or even remove it. Essentially resulting in a dijkstra.
The second reason is that we want to cache all the accumulated data anyway. We are going to run through the entire graph, collecting all distances, and caching them. So that future calls of dijkstra can use the data from previous runs. If we would go over the entire graph with A*, i.e. ignoring the traditional end condition of having the goal in the visited nodes, we would do a whole lot of useless computations to get the heuristics, but these heuristics only bring us faster to the goal. When we decide to ignore the goal in the first place, A* becomes redundant.

2.1 Usage

As stated above we decided to do a Single Source Shortest Distance approach for dijkstra, meaning that we do not care what the goal is, we just start from the source, and traverse the entire graph, caching the distance between any node and the source. Therefor our dijkstra is called like this dijsktra(Source). We also do not expect a result from our dijkstra, since we will cache all the shortest distances. In order to get the shortest distance between two nodes, we would have to do dijsktra from the source, and then query for the distance.

```
dijkstra(Source),
cached_shortest_path(Source, Goal, Distance),!,
member(X, Y).
```

We use cached_shortest_distance because we want to be able to get the shortest path in the same way, even if it is not yet cached.

```
:- dynamic cached_shortest_path/3.
shortest_path(X, Y, D):-
  cached_shortest_path(X, Y, D),
  !.
shortest_path(X, Y, D):-
  dijkstra(X),
  cached_shortest_path(X, Y, D),
  !.
```

Now we can just use shortest_path(Source, Destination, Distance). When the distance is cached, we will retrieve it. And when neccesary we calculate the distance from scratch.

2.2 Optimizations

Since dijkstra is the core of the entire algorithm and is called so very often it is obvious that some optimizations are in order to let the program finish withing a fashionable time window.

2.2.1 Sorting

In the initial naïve implementation we kept track of both the visited nodes ('visited' from now on) and the nodes that are waiting to be expanded ('toVisit' hence forth). When a new node was visited, we would just prepend it to 'visited' and expand its neighbours, wich in turn would them to 'toVisit' when needed. In every iteration of dijkstra we would need the node from 'toVisit' that has the smallest distance to the source. In order to get this node we'd have to sort the list based on the total distance. Which would be in $O(n \log n)$ (At least thats what we would expect from a build in implementation of sort). This means that for every node that we visited we need to sort the list at least once. Making our overal performance in $O(n^2 \log n)$. Not very fast for the core of the algorithm.
In the updated version we tried to get rid of the sorting in each iteration by keeping 'toVisit' in a ordered list. We used the 'ordset' package from prolog for this. This means that inserting an element in the list would be in $\log n$, doing this for each node we visit this would bring give us a total of $n \log n$. This seemed reasonable, and

test cases for our test graph showed that we could get the distance from one node, to all the nodes in the graph under 1 second. The use of a set was valid in this setting since we are not allowed to have the same node twice in 'toVisit' anyway. If we would have this, one of the distances would be smaller (or they would be equal, in wich case it wouldnt́ matter anyway), so we get rid of the larger of the two, resulting again in an ordered list of unique elements.

2.2.2 Caching

When we are asking for the shortest distance between two nodes, we choose to get all the distances between the source and every node in the graph. Doing this would mean very little if we didnt́ use this information for future calls of dijkstra. Therefor, when the dijkstra is finished, and we have list of visited nodes, we can traverse this list, and insert an entry in our knowledge base for each of them. The next time we ask for this distance, we can just query for it.

```
1  dijkstra(X):-
2    list_to_ord_set([(0, X)], Q),
3    dijkstra_1(Q),
4    cache_distances(X) -> true;
5    retractall(visited(_, _)).
6
7  cache_distances(X):-
8    visited(Distance, Node),
9    assert(cached_shortest_path(X, Node, Distance)),
10   fail.
```

2.3 Using the knowledgebase

As we can see in the code of dijkstra, in the previous section, we let the dijkstra run with an singleton initial set, (containing the source), we then cache all the results we found. After all this is done we do a 'retractall(visited(_,_)).'. This is because of another optimization we used.
When we run our dijkstra, we check for each node we expand, if we have already visited it. If we have we ignore the new node, if not we enqueue it in our 'toVisit' set. However, checking if we already visited the node would mean we have to traverse our 'visited' list, and check if the node is a member of this list. This is also expensive if we have a big visited list(and we asume we do), this would be in $O(n)$ and we do this once for each node, result in $O(n^2)$. This is once again to slow for the core of the program. And we decided to change the way we check if a node is visited, by using the prolog knowledge base. Every time we would add a node to the visited list, we instead assert the node as visited, and keep track of the distance we calculated.

```
1  dijkstra_1([(Distance_to_node, Node)|Q]):-
2    ...
3    assert(visited(Distance_to_node, Node)),
4    ...
5    dijkstra_1(New_Q).
```

And then instead of looping over the list to check if the node is visited, we just query for it.

```
1  queue_node(Node, _, Q, Result):-
2    visited(_,Node),
3    Result=Q,
4    !.
```

2.4 Missing optimizations

All these optimization resulted in a decently fast dijsktra that could get the shortest distance between two nodes in under a second (in our test graph). However there is still one optimization we liked to have implemented to further speed up the process.
When we do a dijkstra from 'Source', we eventualy traverse the entire graph, resulting in cached values for all distanced starting from 'Source', to any other node. However, we calculated so much more that we are not caching. And since we use dijsktra all the time, we are essentially doing to much work on future passes.
Our caching mechanism works as follows: our dijkstra collects all distances. We loop over all the collected data, and assert it in the knowledge base.

```
cache_distances(X):-
  visited(Distance, Node),
  assert(cached_shortest_path(X, Node, Distance)),
  fail.
```

Supose we have three nodes A, B and C. And we query for the shortest distance between A and B. This would then traverse the graph, and cache all distances from A to any node. Supose that the shortest path that exists between A and B is [A, A1, A2, ..., B] and the shortests path between C and B is [C, A1, A2, ..., B] in other words, the two paths have all nodes in common, exept their start node. We would like to have cached the distance betwee A1 and B, and every intermediate distance as well for that matter. But unfortunatly we dońt. For our second query (the distance between B and C) we only benefit all paths that go through A. So we are doing to much work on consecutive passes through the graph.