ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Computer Science Master, Course: Intelligent Agents
Students: Ivan Gavrilović, Boris Perović

Assignment 4

Programming: Implementing a Centralized Agent

A brief solution description:

The solution was developed in IntelliJ IDEA 12 IDE, using JDK 7 update 40 and the provided LogistPlatform framework. Git was used as collaboration tool and a version control system. Provided template (*centralized.zip*) was used as a base for the project and agent centralized-random given in it was used for comparison purposes.

The provided solution of the problem that allows one vehicle to carry only one task at the time was used as a foundation for developing algorithm in which vehicles can carry multiple tasks at the same time.

Challenge was to define precisely the set of variables, set of domain values for variables, set of constraints and the objective function. After this was done, Stochastic Local Search algorithm for COP was used to find the sub optimal solution of the problem. The details of the implementation will be discussed in this report as well as the obtained results.

Set of variables:

CSP was defined using `vehicleAction` structure in class `A`. The pickups or deliveries of the tasks are considered as separate `Action` objects (`Action` objects contain a reference to the `Task` object and the type of the action, pickup or delivery). Map contains vehicles as keys, and for vehicle `v`, `vehiclesAction[v]` contains list of all actions that should be executed by `v`. Execution order of actions is specified by list as it maintains the order of all added elements. Each vehicle has its own entry in this map. Value for any key is either empty list, meaning that `v` does nothing, or list containing some actions.

Set of domain values for variables:

Representing the task manipulation actions at the right granularity was vital in order to express all possible outcomes. Therefore, we define set of `DomainVal.actions` that contains all possible task pickup and delivery action. There are `2*t` values in `DomainVal.actions`, where `t` is the number of tasks. We define and `DomainVal.vehicles` which contains all the possible vehicle values. Every element form `DomainVal.vehicles` has an entry in `vehicleAction` map. Values from `DomainVal.actions` are the only ones that can be found in the list of actions for any vehicle in the `vehicleAction` map.

Set of constraints and objective function:

- For vehicle `v`, `vehicleActions[v] = (`a_1`, `a_2`,...)`, then a_1 is pickup action.

- a_i, a_j are pickup and delivery of `Task t`, respectively. Then for some vehicle `v`, we have `vehicleActions[v] = (..., ai, ..., aj, ...);` both actions must be executed by the same vehicle and pickup must be before delivery.

Every change of assignment checks against capacity constraint that is defined in `changingActionOrder` in `SLS`.

Objective function is almost the same as from the paper; see cost method in `A` class.

SLS – Initial solution:

We chose to initially distribute each task to the vehicle whose cost from home city to pickup city of the task is the lowest. In each loop iteration one task `t` is assigned to the nearest vehicle `v`, whose capacity allows to pickup `t`. Pickup and delivery action for task `t` are added for `v` in the `vehicleActions` map. If there is a task whose weight exceeds the maximum capacity of the biggest vehicle, error is generated.

SLS – Choose neighbours:

Structure is the same as the one in the pseudo code from the paper. Here will be explained *ChangingVehicle* and *ChangingTaskOrder* operations.

- *ChaningVehicle* – method `changingVehicle` in `SLS` class implements this. The first action for the randomly selected vehicle v_i is selected. This action is certainly pickup action for some task `t` (constraints specify that). The corresponding delivery action for task `t` is found in the list of actions, and removed. Both of these actions now become the last actions executed by a new vehicle, and corresponding list of actions for vehicle v_j is updated. Before invocation of this method, we check if the capacity of the new vehicle can handle task `t`.
- *ChangingTaskOrder* – this is implemented in `changingActionOrder` method in `SLS` class. The two actions may change places only if none of the constraints are violated. This swap is invalid if it puts delivery for some task `t` before pickup of task `t`. Also, the requirements for the capacity may change when we reorder the tasks, so this is checked as well. This changes list of action for vehicle.

SLS – Local choice and termination condition:

Local choice operation first chooses the assignment with the best cost amongst the neighbours (if there are multiple with the same cost, picks a random one). Based on this result, the global best assignment found so far is updated if needed. By best assignment, we refer to the one with smallest cost. Nonetheless, the operation returns old solution with probability `P` and the new one (chosen amongst neighbours) with probability `1-P`, as explained in the paper. `P` was set to 0.5, as suggested in the paper.

The algorithm terminates after specified number of iterations, and we use the global best assignment to generate the plans for each of the vehicles afterwards.

Obtained results:

One initial solution implementation was previously mentioned. It is distribution of the task to the vehicle with the smallest cost from home city to pickup city (*nearestToHomeCity*).

Whether the results change if the initial solution changes can be inferred from three tables below. *allToBiggest* assigns all tasks to the biggest vehicle (if there are more than one, chooses the one with biggest `id` value), and *assignRandom* randomly assigns tasks to the vehicles. All three methods were used to generate initial solution. Assignments from *nearestToHomeCity* and *assignRandom* were using in SLS terminating after 50000 iterations, while *allToBiggest* had 5000 iterations. Tests were run for 3 and 5 vehicles for 30 tasks, using England map. *nearestToHomeCity* and *allToBiggest* are somewhat comparable, while *assignRandom* clearly has the worst performance. Top row indicates the cost of the initial solution generated by specified method, while bottom row contains cost of the plan after the SLS.

	nearestToHomeCity | 3 vehicles					*nearestToHomeCity* | 5 vehicles				
Start	71593	62569	71262	76515	83540	69230	60668	65990	76080	82980
End	15255	19412	21253	20481	18465	20099	22885	20822	21258	17718

Tb1 - nearestToHomeCity for inital solution

	allToBiggest | 3 vehicles					*allToBigges* | 5 vehicles				
Start	68731	65364	68681	74894	76809	68731	65364	68681	74894	76809
End	17370	17344	19765	17487	17919	16627	14052	19175	15196	16769

Tb2 - allToBiggest for inital solution

	assignRandom | 3 vehicles					*assignRandom* | 5 vehicles				
Start	76506	71672	72646	79553	77656	71615	73336	73551	77667	82730
End	24949	29875	28793	23648	29567	31374	36813	40902	35115	39360

Tb3 - assignRandom for inital solution

Naive plan gives the cost around 68731 for both 3 and 5 vehicles, as it uses only one for all tasks.

The centralized agent implementation improves the cooperation between agents. Deliberative agent implementation from previous project had a weak point when multiple agents were run. Collisions occurred and plans had to be recalculated, and all that increases the costs for the company. Below, we can see the two implementations run with the 13 tasks, using England map. The centralized agent is a clear winner here. Vehicles have plans that do not interfere and that bring more profit to the company.

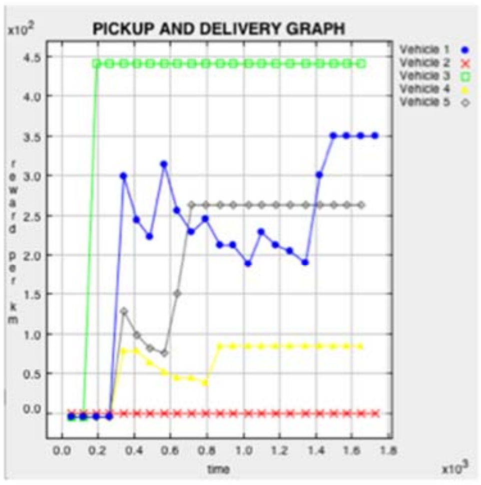

Chart 1 – centralized agent reward per km

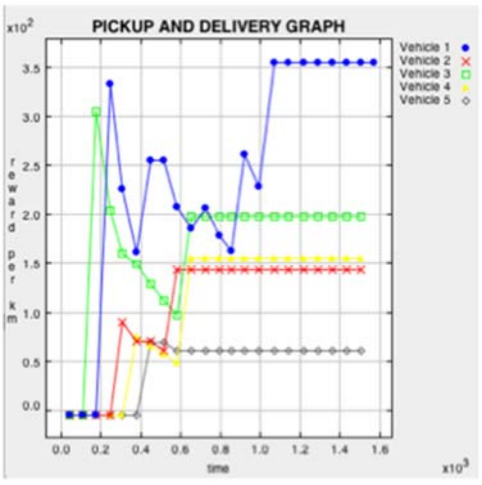

Chart 2 – deliberative agent reward per km