Excercise 4
Implementing a centralized agent

Group №76: Simon Honigmann, Arthur Gassner

November 6, 2018

1 Solution Representation

1.1 Variables

A `Solution` class was created to better store and manage information for each potential delivery solution. A `Solution` object consists of a `HashMap` of `Vehicles` as key and `ArrayLists` of `Actions` as value, also called an agenda. A reduced version of each agenda, which simply includes information for whether the task is to be picked up or delivered and where, in the order in which the vehicle is supposed to perform those actions is also stored in the `Solution`. Finally, the texttttSolution has a method, which can convert the simplified list of pick-ups and drop-offs into a complete action list suitable for Plan generation.

1.2 Constraints

Relatively few constraints are applied to the solution representation. Constraints include the following:

1. Solutions cannot result in a vehicle carrying more tasks that its capacity allows.
2. Deliveries must always be after pick-ups in the generated task lists.
3. A vehicle which picks up a task must also deliver that task. There is no exchange between vehicles, within a plan.
4. All tasks provided upon initialization must be picked up and delivered in the generated solution.
5. A solution must be generated within the time-out specified in the default_settings.xml file.

1.3 Objective function

The function to be optimized is the sum of the costs incurred by each vehicle for a given solution. This allows each solution's optimality to be comparable.

2 Stochastic optimization

2.1 Initial solution

Several methods for generating an initial solution were explored. First, a naive approach which sequentially distributed tasks between each vehicle, to result in a distributed initial task allocation. Second was a greedy approach, which assigned every single task to the first vehicle in the vehicle list which could carry the task. This resulted in the first vehicle having the majority of tasks assigned to it. It is assumed for all initial solutions, that at least one vehicle can carry the largest task in the specified task list. If a task is ever larger than the selected vehicle's capacity, it is assigned to a different vehicle until the selected vehicle has a large enough capacity.

Ultimately, our testing showed that the naive approach was the most consistent with a low total final cost. This initialization was therefore used in all future experiments.

2.2 Generating neighbours

First a vehicle that has at least 1 task is randomly selected. Then each task carried by the chosen vehicle is reinserted into for each possible location of the task pick-up and task delivery in every vehicle (including the chosen vehicle). Each solution represented by the new task location is set as a new neighbour. Every generation created by this method therefore contains neighbours created by swapping task positions within a given vehicle and by transferring tasks between vehicles. Only neighbours that satisfy the constraints listed above are returned. The neigbours generation therefore works in $O(number\_of\_vehicles * (number\_of\_tasks)^3)$.

2.3 Stochastic optimization algorithm

Every time neighbours are generated, the local minimum is first found and compared to the current best solution. If it outperforms the current best, the current best is replaced by the local minimum. To implement the stochastic local portion of stochastic local search, two probability thresholds are defined: `P_Upper` and `P_Lower` which determine how the next iteration's base solution will be selected. Every iteration, a random number is generated between 0 and 1. If the number is below `P_Lower`, the local minimum solution is returned and used to generate the next set of neighbours. If the number is between `P_Upper` and `P_Lower`, inclusive, the previous solution is used to generate neighbours. If the number is above `P_Upper`, a random solution is selected from the set of neighbours and is used to generate future neighbours.

To formalize our approach, we added local-minimum loop detection. If the optimal solution stayed the same for more than `MAX_REPEAT` times, then a random neighbour is automatically selected. This is to try to avoid getting caught at local-minima, without sacrificing performance early in the search by selecting random neighbours when the new local minima are consistently improving. After adding this, `P_Upper` was set to 1 so that random selection is only used when solution stagnation occurs.

3 Results

For all experiments, plan time out was set to 30 s and every value reported was averaged over 3 trials.

3.1 Experiment 1: Model parameters

In the first experiment we tested the impact of texttttP_Lower and `MAX_REPEAT` on the final plan cost.

3.1.1 Setting

Settings for this experiment are specified in `E1_centralized.xml` and `E1_settings_default.xml`. This experiment always used 30 tasks, 4 vehicles, and the England topology. Values of `MAX_REPEAT` of 2, 10 and 20, and `P_Lower` of 0.1, 0.5, 0.8 and 1 were tested. The results are included in the following section.

3.1.2 Observations

As is shown in the table, `MAX_REPEAT` and `P_Lower` strongly influence the performance of the solution generation. When `P_Lower` is too small, the search is too likely to hold on to old solutions and does not explore enough to find optimal solutions. As `P_Lower` increases, the solution favours local minima and stops holding onto old solutions. There also appears to be an optimal number of allowed repetitions during loop checking. Too few and the search wanders in random directions too often. Too many, and the algorithm gets stuck at local minima without exploring enough to find more optimal solutions.

Table 1: Plan Cost for Different `MAX_REPEAT` and `P_Lower` Values

MAX_REPEAT	P_Lower			
	0.1	0.5	0.8	1.0
2	$42,815	$26,079	$19,338	$19,806
10	$24,552	$17,556	$16,087	$15,268
20	$21,051	$17,251	$15,724	$18,158

3.2 Experiment 2: Different configurations

In this experiment, we wanted to compare the efficiencies of different numbers of vehicles to see how evenly distributed tasks were among the vehicles.

3.2.1 Setting

Settings for this experiment are specified in `E2_centralized.xml` and `E1_settings_default.xml`. We run the experiment for 1, 4 and 10 vehicles, each time with 5 and 30 tasks.

3.2.2 Observations

Table 2 shows the cost for different plans, as well as the number of vehicles actually carrying tasks (VCT) for each plan. As can be seen from the data in Table 2, for a small number of tasks, there is little benefit of having multiple vehicles. There is also a high disparity among vehicles in terms of number of tasks carried. One vehicle tends to carry all or the majority of the tasks for the group. As the number of tasks increases, the task distribution also improves, and more vehicles have tasks to carry. However, it is still almost always found to be more efficient for most vehicles, as seen with the 10 vehicle data points, to sit idle. We would expect a very different distribution of labour if the optimization function included a time cost, preferring faster solutions to slower solutions. This would result in a more realistic allocation of resources, each vehicle carrying tasks in order to distribute everything as quickly as possible.

Table 2: Plan Cost and Number of Vehicles Carrying Tasks for Different Numbers of Vehicles and Tasks

	Number of Vehicles			
	4		10	
# of Tasks	Cost	VCT	Cost	VCT
5	$5,752	1	$5,752	1
30	$16,966	2.33	$16,625	3

The complexity of the algorithm grows quickly with the number of tasks, as it generates neighbours for every possible pick-up and drop-off action positions in the recipient vehicle's task list. This results in considerable growth in complexity with respect to the number of tasks. The growth is proportional to the number of vehicles. If a very large number of tasks were to be computed, the algorithm could be modified to only generate a limited number of random neighbours rather than all possible neighbours.