Ecole Polytechnique Fédérale de Lausanne - EPFL

CS-430 Intelligent Agents

# Mission 4

**Centralized Agent**

Vladimir Somers
Armand Bosquillon de Jenlis

Group 27

*Professor* : Boi Faltings

Academic Year 2015 - 2016

## 1 Introduction

In this report, we will first talk about the design choices we made and the new representation of the CSP. We will then present the main algorithm and the improvements we made to it. We will finally present the results of different strategies we implemented.

## 2 Design Choices

Our design is very similar to the one explained in the document pdp_as_csp. In order to allow the vehicles to transport multiple tasks at a time, we had to do some modifications that are presented below. Here we just explain the changes we made.

First we define a few constants,

- **Nt** is equal to the number of **tasks**,
- **Na** is equal to **2*Nt** and is the number of possible **actions** (explained later).
- **Nv** is equal to the number of **vehicles**.

### 2.1 CSP representation

In our new representation, we add the fact that we can make two actions: **pickup** a task or **deliver** a task. Thus it is no more mandatory to deliver a task just after it has been picked up.

- We represent each task **t** by a number in $[0, 1, ..., Nt - 1]$.
- We have two type of actions **a**, either it is a pickup **P** action or a delivery **D** action.
- We represent the pickup action **P** with a number in $[0, 1, ..., Nt - 1]$. If **P** = **t** then we pickup task t.
- We represent the delivery action **D** with a number in $[Nt, Nt + 1, ..., Na - 1]$. So if an action **D** is such that **D-Nt** = **t**, then we it's a Delivering of task t.

In order to transport multiple task we use a table **load**. This table has **Na** entries.
For each action **a**$\epsilon[0, 1, ..., Na - 1]$, **load(a)** = the load of the vehicle which executes the action **a**, just after it has executed it.
We added the table **previous** which is similar to the table **next** except that **previous(a)** = **a'** means that some vehicle will make action a' and then action a. We created this table for the implementation to be easier and to have a faster search.
We also had to define some new **constraints**:

- load(a) $\leq$ vehicle(a).capacity()
- For each pair **P, D**, such that $P$ = *D-Nt*, we have to be sure that **time(P)** $<$ **time(D)**.

  For now when we talk about a task, we often mean such a **pair P, D**.

## 3 SLS

Our implementation of the SLS follows the one given in pdp_as_csp. In order to have a better solution, we added a new loop which executes the SLS algorithm with 2*Nv different initial solutions.

### 3.1 SelectInitialSolution(int index)

This method create two type of initial solution depending on the value of its argument. At the beginning of the method, we first check that no task has a weight bigger than the capacity of the vehicle that has the biggest capacity. For the two types of solution, each time the weight of a task is bigger than the capacity of the vehicle that should carry it, we put the task on the vehicle with the biggest capacity. The two solutions are the following:

1. If index $<$ Nv, then for each task, try to put it in vehicle(index).
2. If index $\geq$ Nv For each task, try to assign it to a random vehicle.

The 2*Nv initial solutions that we create are first, for every vehicle, to try to put the task in it and secondly, try Nv times a randm assignation of tasks.

## 3.2 ChooseNeighbours

We made two changes in this method.
First, the way we try to move a task from a vehicle to another vehicle. As before, we pick a random vehicle v, then for each task (pair P, D) of **v**, for each vehicle **v'**, we try to put the task in every possible position of the task list of **v'**. This means that for every corresponding pair P, D in the action list of v, we try for every other vehicles v' to put these pairs in every possible positions in the action list of these vehicles.

The second important change is that we keep in a global variable the best solution that we found so far. This best solution is updated each time a newly created neighbor is better than the previous global best solution, even if we don't explore this neighbor later in the algorithm.
These improvements to the *ChooseNeighbours* function is the most important change we made regarding the performance of the search, because this new function generates way more neighbors than the previous one and allows to travel the solution space in a more efficient way. With this method, we get results with half of the cost of the results we had with the previous one!

The change of the task order is the same as the one described in the given document.
This method executes O($Nt^3 + Nv$) times the method ChangingVehicle then O($Nt^2$) times the method ChangingTaskOrder thus it executes in $\mathbf{O(}Nt^4 + Nv * Nt\mathbf{)}$.

## 3.3 LocalChoice

In order to choose the next solution, we first put all the neighbors and the current solution in a list L. Then we choose the new solution A as follow:

- with probability 0.7, we set A to the best solution of L.
- with probability 0.3, we set A to a random solution in L.

## 3.4 ChangingVehicle(NodePD A, int v1, int v2, int aIndex, int pIndex, int dIndex)

This function moves the task which is picked up by v1 in position aIndex of its action list and put it in the action list of v2. It puts the pick up action in position pIndex and the delivery action in position dIndex. It first checks that moving the actions will not violate the two constraints

- load(a) $\leq$ vehicle(a).capacity()
- for each pair **P, D** such that P = D-Nt we have to be sure that **time(P)** $<$ **time(D)**.

Then it moves the actions.
This function is executed in **O(Nt)**.

## 3.5 ChangingTaskOrder

This method receives two actions a1 and a2. It first checks if by changing the order of a1 and a2 in the action list of their vehicle, we do not violate the two constraints

- load(a) $\leq$ vehicle(a).capacity()
- for each pair **P, D** such that P = D-Nt we have to be sure that **time(P)** $<$ **time(D)**.

Finally if these constraints are not violated, it changes the order of a1 and a2.
This function is executed in **O(Nt)**.

## 3.6 Properties of SLS

Our search is stochastic because there's some randomisation in the way the next solution to explore is chosen in the SLS algorithm and in the way we generate the neighborhood of the current dol. This stochastic approach is important to avoid falling in a local minima.
Our search is Local because: at each iteration of the search, we go from a solution to an other solution which is very close to it (very similar).

### 3.7 Stopping criterion

We try 5000 iterations for each different initial solution. There's also a condition which checks that there's no time-out : we return the best solution found so far if the elapsed time is at 95% of the allocated time.

### 3.8 Fairness of the solution

In most cases, the solution is not fair in the sense that the tasks are not equivalently spread between the vehicles. This observation is normal because, on a map with a lot of tasks, the utility of one vehicle is greatly increased if it is fully loaded. This is why increasing the number of vehicle above a certain threshold is often useless : these vehicles will not help to find a better plan because an optimal plan often involves few vehicles carrying as much tasks as possible.

## 4 Simulation Results

We can observe, thanks to the simulation results, that our search is stochastic as we almost always get different results when we launch the same search, with the same number of iteration, on the same initial solution. We also present, in the annexes, some timing comparison when increasing the number of vehicles and the number of tasks.

### 4.1 Comparison of different strategies

In order to compare the different strategies, we create the following table that reports for multiple strategies, 4 different final solution results. These results are computed with the map England, 4 vehicles and 30 tasks. The best implementation tries 5 random initial solution with the extended neighbour function and the LocalChoice described in the report. The "*Old neighbour function*" is the classical strategy (given in the document) with the multiple task hypothesis added. The "*One task*" strategy is the naive strategy given in the document.

| Best implementation | | | |
|---|---|---|---|
| 12445 | 12932 | 12914 | 11502 |
| Old neighbour function | | | |
| 31389 | 32772 | 30678 | 33457 |
| OneTask | | | |
| 50541 | 50541 | 50541 | 50541 |

Table 1: Comparison of strategies

### 4.2 Particular situations

If we are in a situation where there is one vehicle v1 that has a lower cost per kilometers than the others and a vehicle v2 that is the only one able to carry some task t, then we can observe that v1 will deliver every task except t and v2 will deliver t. This proves that our centralized agent is clever.

## 5 Conclusion

The centralized agent brings a new efficient approach to compute an agent's plan when there are more than one vehicle involved. From the given template, there were few changes who changed dramatically the efficiency of the search :

1. The possibility for a vehicle to carry multiple tasks.
2. The way we keep track of the best solution we found.
3. The computation of larger neighborhood of a solution.
4. The multiples restarts of SLS with different initial solutions.
5. The new stochastic local choice.

## 6 Annexes

### 6.1 Impact of the number of tasks on the execution time.

As we can see when we augment the number of tasks the execution time augments quickly. The tests were run with the best strategy on England with 4 vehicles and 5000 iterations. The ti

| Number of tasks | 10 | 30 | 50 |
|---|---|---|---|
| Execution time (s) | 1.8 | 12.4 | 68.1 |

Table 2: Execution time comparison for task number

### 6.2 Impact of the number of vehicle on the execution time.

As we can see when we augment the number of vehicle the execution time augments slightly. The results are given for 30 tasks, in England and with 5000 iterations.

| Number of vehicles | 1 | 3 | 4 |
|---|---|---|---|
| Execution time | 10.2 | 12.1 | 12.8 |

Table 3: Execution time comparison for vehicle number