Traveling Salesman Problem with a Drone – CPLEX model

https://www.andrew.cmu.edu/user/vanhoeve/papers/tsp_drone_cpaior2019.pdf

Chile Ovidiu-Benone MOC2

Pintilie Andrei MOC1

The Traveling Salesman Problem with a Drone (TSPd) is an extension of the Traveling Salesman Problem (TSP – also known as Traveling Salesman Person). In the TSP problem, a truck has to deliver a parcel to n houses, and must do it so by only visiting each house once. Basically, it is the same as finding the shortest Hamiltonian cycle of an undirected graph.

The TSPd problem introduces the concept of a drone which helps with the delivery: the drone can get a single parcel from the truck and deliver it to one of the houses; after doing so, it must go back to the truck to get another parcel. This synchronization between the truck and the drone can only be done at one of the house.

The article at https://www.andrew.cmu.edu/user/vanhoeve/papers/tsp_drone_cpaior2019.pdf provides a solution to this problem using the Constraint Programming (CP) method. Although the CP model given could be better detailed, the authors (Ziye Tang, Willem-Jan van Hoeve1, and Paul Shaw) defined and proved the formalities behind it, providing also a complexity analysis and some examples. We have adapted the pseudocode for the CP model to use it in IBM's tool CPLEX (version 12.9.0).

Our problem, TSPd, is also known as FSTSP (the drone is named a Flying Sidekick, hence the **FS**TSP), UAV (Unmanned Airborne Vehicle) routing, and many other similar names.

From pseudocode to CPLEX OPL code

Having the CP pseudocode, it was easy to create the OPL code:

```
minimize
     endOf(tVisit[n]);
subject to {
     forall (i in Cities) presenceOf(visit[i]);
     presenceOf(tVisit[0]);
     presenceOf(tVisit[n+1]);
     first(tVisitSeq, tVisit[0]);
     last(tVisitSeq, tVisit[n]);
     noOverlap(tVisitSeq, w);
     noOverlap(dVisitSeq, w);
     forall(i in Cities) {
           alternative(visit[i], append(tVisit[i],dVisit[i]));
           alternative(dVisit_before[i], all (j in Cities) tdVisit[i][j]);
           alternative(dVisit_after[i], all (j in Cities) dtVisit[i][j]);
           span(dVisit[i], append(dVisit_before[i], dVisit_after[i]));
           endAtStart(dVisit_before[i], dVisit_after[i]);
           presenceOf(dVisit[i]) => presenceOf(dVisit_before[i]) &&
           presenceOf(dVisit_after[i]);
     }
     forall(i, j in Cities) {
           presenceOf(tdVisit[i][j]) => presenceOf(tVisit[j]);
           presenceOf(dtVisit[i][j])=> presenceOf(tVisit[j]);
           startBeforeStart(tVisit[j], tdVisit[i][j]);
           startBeforeEnd(tdVisit[i][j], tVisit[j]);
           endBeforeEnd(dtVisit[i][j], tVisit[j]);
     }
}
```

The problem is shaped as a scheduling problem, where we have the following intervals:

- *visit[i]* – the time at which the drone or the truck arrives and leaves node i
- *tVisit[i]* – the time at which the truck arrives and leaves at node i
- *dVisit[i]* – the time at which the drone arrives and leaves at node i
- *dVisit_before[i]* – the time at which the drone arrives at node i
- *dVisit_after[i]* – the time at which the drone leaves node i
- *tdVisit[i][j]* – the time for the drone to leave the truck at node j and goes to node i
- *dtVisit[i][j]* – the time for the drone to leave at node i and return to truck at node j

All intervals are marked optional as not every interval needs to be present, apart from the *visit* which is forced to be present.

As more informally explained, the constraints are:

- *Line 4* – All nodes needs to be visited
- *Line 5->8* – The start and end node needs to be visited. The start needs to be the first and end needs to be the last.
- *Line 9->10* – truck visits have to be at least w_{ij} apart if both of them are served by the truck, similarly for the drone.
- *Line 12* – a node can be visited by truck or drone
- *Line 13->14* – drone arrives at a node from truck and leaves node towards the truck
- *Line 15->18* – drone visit consists in the time it arrives and the time it leaves
- *Line 21->25* – if truck and drone meets at a node, the node must be visited by the truck and the time it stays at a node is enough to sync with the drone

In current testing the following model returns a solution, but it is not the optimal solution.

As a result this model will serve us for continuing solving the problem.

Linear programming

After trying many ways to make the CPLEX code work, we have searched in many other articles for a (Mixed) Linear Integer Programming model to use for our cause. The best one was:
https://pdfs.semanticscholar.org/1112/cb5d327e4cd50b07e353298ca5c0aa094bb5.pdf

$$\min \sum_{o \in O} c_o x_o \tag{13}$$

$$\text{such that} \sum_{o \in O(v)} x_o \geq 1 \qquad \forall v \in V \tag{14}$$

$$\sum_{o \in O^+(v)} x_o \leq n \cdot y_v \qquad \forall v \in V \tag{15}$$

$$\sum_{o \in O^+(v)} x_o = \sum_{o \in O^-(v)} x_o \qquad \forall v \in V \tag{16}$$

$$\sum_{o \in O^+(S)} x_o \geq y_v \qquad \forall S \subset V \setminus \{v_0\}, v \in S \tag{17}$$

$$\sum_{o \in O^+(v_0)} x_o \geq 1 \tag{18}$$

$$y_{v_0} = 1 \tag{19}$$

$$x_o \in \{0, 1\} \qquad \forall o \in O \tag{20}$$

$$y_v \in \{0, 1\} \qquad \forall i \in V \tag{21}$$

Unfortunately, because of the difficulty to model subsets of different sets, this model was not further used.

The Multiple Flying Sidekicks Traveling Salesman Problem (mFSTSP)

While searching for alternatives for the CPLEX pseudocode given in our article, we have stumbled upon a Python implementation of the mFSTSP problem. This problem practically introduces m drones instead of only one (the implementation should also solve the problem with many trucks too, but we don't need this). If we fix $m = 1$ we basically have a solution for our own problem, TSPd.

The authors, Chase C. Murray and Ritwik Raj (University of Buffalo, USA), have formally described their methods in the article The Multiple Flying Sidekicks Traveling Salesman Problem: Parcel Delivery with Multiple Drones, July 27, 2019.

Python code has been given at https://github.com/optimatorlab/mFSTSP, which can solve the problem in one of two ways: either by using an own heuristic which combines 3 different subproblems, either by using MILP with the Gurobi Solver. Because the Gurobi Solver requires an academic license, which can only be activated from within the university network, we couldn't use it (yet). This means we could only use the heuristic method.

The downside of using this solution is that it has much more complicated input to give, with many parameters which need to be specified.

The input contains:

- Node ID – just an index
- Node type – if the node is the depot or not
- Latitude Degrees – the location of the house, specified as latitude degrees on Earth
- Longitude Degrees – the location of the house, specified as longitude degrees on Earth
- Altitude Meters – the elevation of the house
- Parcel with lbs. – the weight of the parcel to be delivered at the house (it influences the drone's speed)
- The distance in meters between each two nodes
- The time for the truck to get from one node to another

The parameters include:

- The problem's data directory – obviously needed. This contains info about the input specified above. Here will also be the solutions written
- Vehicle File ID – one of $101, 102, 103$ or 104, for specifying if the speed of the truck or drone will be slow or fast; not needed
- Cutoff time – if the execution exceeds this time, it is halted; not really needed, but useful
- Problem Type – if we use the heuristic method or the Gurobi solver; needed
- Number of UAVs – we need it to be 1
- Number of Trucks – we need it to be 1 (although the solution doesn't support yet more trucks)
- Require truck at depot – if equals to 0, basically specifies if the drone can also supply from the depot, not only from the truck; our problem can use this too, so it is needed
- Require driver – indicates if the drone can launch from and return to the truck while the driver delivers a parcel; we need this to be true, as we must only sync at houses
- EType – Indicates the endurance model that was employed; not needed

- ITER – number of iterations; only used for the heuristic method.

The advantage of this implementation is that if we specify all these parameters and input data as we need, we can solve the TSPd problem we started with. We have only (successfully) ran this Python solution on the inputs given by the authors (they provide a lot of these inputs).

Conclusions

After many attempts to make the CPLEX code work, we have finally arrived at a functional CP model for our problem. The MILP model, although very promising, has failed giving us a model, as we would have needed way too much research to make it functional.

The heuristic method from the authors from the University of Buffalo gives us a robust, fast, but complex solution to our problem, which can still be furthermore used for comparison with other algorithms, like Dynamic Programming or the Branch and Bound method.