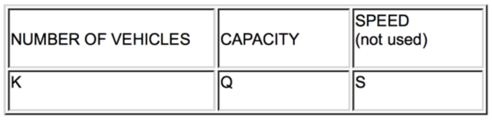
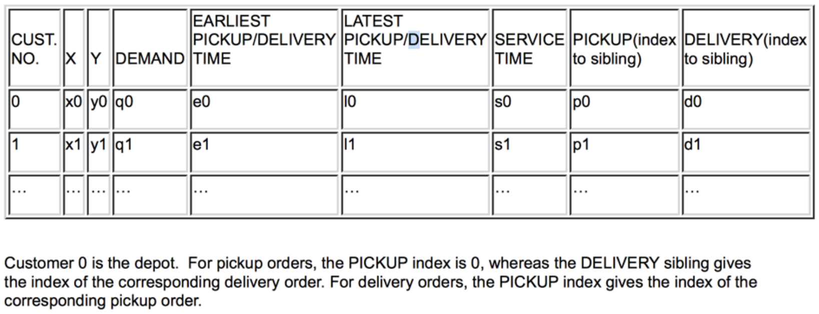
1 Data Generation

To facilitate further testing and to produce computational results, we will generate data following the author’s instruction, with some slight adjustments



NUMBER OF VEHICLES	CAPACITY	SPEED (not used)
K	Q	S



CUST. NO.	X	Y	DEMAND	EARLIEST PICKUP/DELIVERY TIME	LATEST PICKUP/DELIVERY TIME	SERVICE TIME	PICKUP(index to sibling)	DELIVERY(index to sibling)
0	x0	y0	q0	e0	l0	s0	p0	d0
1	x1	y1	q1	e1	l1	s1	p1	d1
…	…	…	…	…	…	…	…	…

Customer 0 is the depot. For pickup orders, the PICKUP index is 0, whereas the DELIVERY sibling gives the index of the corresponding delivery order. For delivery orders, the PICKUP index gives the index of the corresponding pickup order.

> *We used problem instances from those in Li and Lim (2001) for the PDP, which are related to the well-known Solomon instances. The datasets are available at http://www.sintef.no/Projectweb/TOP/PDPTW/*

Although the base problem data is obtainable online, the author has to make changes to the dataset with randomized data generation, here we will quote the data generation procedure from the paper and present our methods of generating data for this problem.

> *We picked subsets of the first 10 nodes and the first 14 nodes to create, respectively, our small 10-node and 14-node problem instances. The pickup-and-delivery requests are paired and so the number of nodes in the network, without the vehicle depots, is even.*

The original dataset does not list pickup and delivery requests in pairs. Therefore, in our implementation, for 10 nodes cases, we simply picked the first 5 pickup and the first 5 delivery requests.

```
    with open(file_name,'r') as f:
        for i,line in enumerate(f):
            # skip the first two lines
            if i == 0:
                speed = float(line.replace('\n','').split('\t')[-1])
                # apprently some files have speed = 0
                speed = speed if speed != 0 else 1
            if i <= 1: continue
            # read lines until pick-up and dilvery information is collected
            cache = [float(piece) for piece in line.replace('\n','').split('\t')]
            # if delivery
            if cache[-1] == 0 and len(delivery) < num_of_customer:
                delivery.append(cache)
            elif len(pickup) < num_of_customer:
                pickup.append(cache)
            if len(pickup) == num_of_customer and len(delivery) == num_of_customer:
                break
```

Here there is some ambiguity in the way the author describes the process. In the Li and Lim (2001) dataset, the demand for each pickup and delivery request are not paired, which means split delivery and pickup is allowed. However, “Splittable pickup-and-delivery requests” variant is only achieved by relaxing $y_{i,j}^{k,r}$ (if the vehicle k carries the request r on the arc ij) from a binary variable to a continuous variable. We are not sure whether the variant is used for the author’s computational tests. As a result, here we will also match the pickup amount and delivery amount.

For the locations of the vehicle depots, we randomly generated origin and final depots for each vehicle, scattered over the X - Y region formed by the nodes of the network. The number of vehicles equals to the number of requests. Therefore, for a 14-node problem instance, seven different vehicles are assigned to seven distinct randomly-generated origin depots and seven distinct randomly-generated final depots.

```
    x_range = [case[1] for case in pickup+delivery]
    y_range = [case[2] for case in pickup+delivery]
    x_min, x_max = min(x_range), max(x_range)
    y_min, y_max = min(y_range), max(y_range)
    start_depot = [(uniform(x_min,x_max),uniform(y_min,y_max)) for _ in
range(num_of_customer)]
    end_depot = [(uniform(x_min,x_max),uniform(y_min,y_max)) for _ in
range(num_of_customer)]
```

We associated a ‘‘cost factor’’ with each vehicle for differentiating the costs of using different vehicles. For each coefficient in the objective function corresponding to each arc in the network, the ‘‘length’’ or the distance *associated with the arc is multiplied by the fractional cost factor of the vehicle to determine the corresponding ''effective'' cost coefficient.*

We will draw from a normal distribution $\sim\mathcal{N}(1,0.1)$ to compute the cost factor. In real life cases, it is reasonable to assume that vehicle with higher capacity will be more expensive to operate. Therefore, we will sort the randomly generated cost factor based on the randomly generated vehicle capacity listed below.

```
    cost_factor = sorted([np.random.normal(1,0.2) for _ in range(num_of_customer)])
    correct_order = sorted(range(num_of_customer), key = capacity.__getitem__)
    cost_factor = [cost_factor[indices] for indices in correct_order]
```

We varied the capacities of the vehicles randomly as well. And, in order to ensure that a problem remains feasible, at least one vehicle is given enough capacity for carrying the largest transport load request.

Without loss of generality, we will fix the last vehicle capacity to be exactly the largest transport load request. Then we will randomly generate vehicle capacity with a normal distribution $\sim\mathcal{N}(\mu(demand), 0.4\mu(demand))$. In addition, we add that the minimum capacity will be equal to minimum demand.

```
        mean_demand = np.mean([case[3] for case in pickup])
        max_demand = max([case[3] for case in pickup])
        min_demand = min([case[3] for case in pickup])
        capacity = [max(np.random.normal(1*mean_demand,0.4*mean_demand),min_demand)
for _ in range(num_of_customer)]
        # fix the last one to be max demand
        capacity[-1] = max_demand
```

---

we adjusted the time windows to only 50% of the originals given in Li and Lim data (2001).

In the original dataset, the time window is only specific to each order and independent of the vehicle. In addition to cutting the time window to half, we also need to compute the travelling time in each arc, by using the speed provided by the original dataset. Here we have to make some adjustments to the author's method, as we have found that it creates infeasibility solutions where delivery time window is ahead of pick-up time window. (e.g. lc101 case request number 4). Therefore, we did an extra step, where we check if the time window is apart with at least two times the minimum transport time.

```
    for _ in range(num_of_customer):
        pickup[_][4] /= 2
        pickup[_][5] /= 2
        delivery[_][4] /= 2
        delivery[_][5] /= 2

    # request time data
    min_tranport_time = [norm(all_depots[j],all_depots[j+num_of_customer])/speed \
              for j in range(num_of_customer)]
    # we need to do a check if the time window is consistent (min time is left)
    arrival_time = [case[4] for case in pickup]\
    + [max(case[4],2*min_tranport_time[i]+pickup[i][4]) for i,case in
enumerate(delivery)]
    depart_time = [case[5] for case in pickup]\
    + [max(case[5],2*min_tranport_time[i]+pickup[i][5]) for i,case in
enumerate(delivery)]
```

For proper comparison of the computational results, the same vehicle depot locations are maintained for two different computational runs: the problems are first solved without transshipment, following which they are solved again with the same dataset but allowing transshipment at all nodes in the network.

Here we will consolidate all the data we have pre-treated and write a .dat file for modelling, note that the cost and time data are computed on the fly.

```
    os.makedirs('./processed_data/',exist_ok=True)
    with open('./processed_data/'+file_name.split('/')[-
1].replace('.txt','.dat'),'w') as writer:
        # set of all nodes
        writer.write('set N := ')
        writer.write(' '.join(double_letter_set[:4*num_of_customer]))
        writer.write(';\n\n')
        # set of transhipment nodes, exclude start and end depot
        writer.write('set T := ')
        writer.write(' '.join(double_letter_set[:2*num_of_customer]))
        writer.write(';\n\n')
        # vehicle data
        writer.write('table\tK={K}\tu(K)\to(K)\to_(K):\n')
        writer.write('\tK\tu\to\to_\t:=\n')
        for k in range(num_of_customer):
            writer.write('\t{:d}\t{:.2f}\t{:}\t{:}\n'
                         .format(k+1,capacity[k],double_letter_set[2*num_of_customer+
2*k],double_letter_set[2*num_of_customer+2*k+1]))
```

```
        writer.write(';\n\n')
        # cost data
        writer.write('param c default 0 :=\n')
        for i in range(4*num_of_customer):
            for j in range(4*num_of_customer):
                if i == j: continue
                if i >= 2*num_of_customer and j >= 2*num_of_customer: continue
                for k in range(num_of_customer):
                    distance = norm(all_depots[i],all_depots[j])
                    writer.write('{:}\t{:}\t{:d}\t{:.2f}\n'
                                 .format(double_letter_set[i],double_letter_set[j],k+
1,distance*cost_factor[k]))
        writer.write(';\n\n')

        # request data
        demand = [case[3] for case in pickup]
        pickup_time = [case[4] for case in pickup]
        writer.write('table R={R}  q(R)  p(R)  d(R):\n')
        writer.write('\tR\tq\tp\td\t:=\n')
        for i in range(num_of_customer):
            writer.write('\t{:}\t{:}\t{:}\t{:}\n'
                         .format(i+1,demand[i],double_letter_set[i],double_letter_set
[i+num_of_customer]))
        writer.write(';\n\n')
```

The above code is for data without time window. The data with time window is generated similarly and will not be listed here.