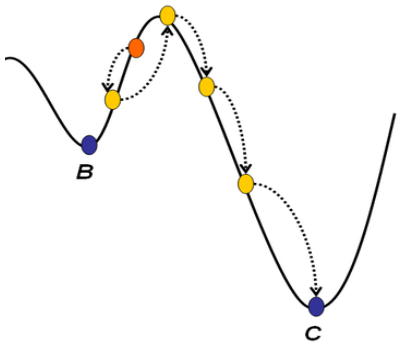
# Programming Exercise 3: Centralized Coordination Model Solution

# 1. Conceptual overview of solution

# Considerations for any solution

- **Representation:**
  - All relevant information about the complete solution and plan for all vehicles.
- **Operators:**
  - Local variation
  - Sufficiently small steps
  - All possible solutions much be reachable via operations
- **Optimization:**
  - Possibility of exploration even in a case of local optimum



# Model solution overview

- The algorithm described in the paper modified to accommodate carrying multiple tasks
  - Simplified but equivalent representation of solution
  - Modified changing task order operator

# Primary changes in comparison to paper

- **PDAction:** Task + pickup flag
- Changes in **ChangingVehicle**:
  - Remove both PDActions corresponding to task pickup and task delivery from V1
  - Place PDAction for pickup and PDAction for delivery at the beginning of V2's plan
- Changes in **ChangingTaskOrder**
  - Instead of swapping places of two tasks, the pickup and delivery locations of a chosen task are changed in a plan

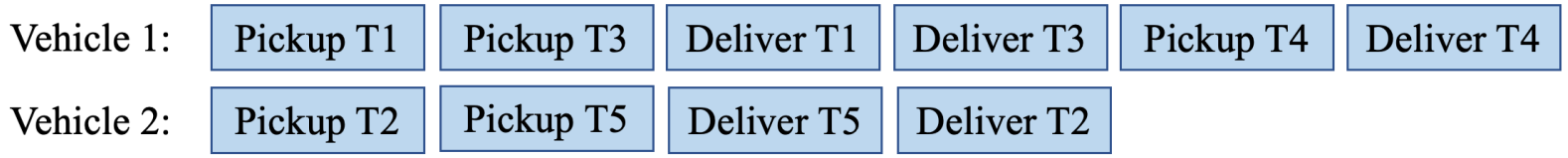
# 2. Overview of implementation

# Files

- Three files:
  - **CentralizedTemplate:** SLS Algorithm and plan generation
  - **Candidate:** Class that represents a candidate solution, plus operators on it
  - **PD_Action:** Task + pickup flag

# Candidate: Representation of a solution

- Main information represented in "plan”
  - List of sublists for each vehicle
  - Each sublist is the plan of vehicle: PDActions in order




- Helper variables “taskList”, “vehicles”
- Cost in “cost”

# SLS algorithm overview

1. Generate neighbours
2. Choose best neighbour with stochasticity
3. Terminate when timeout is reached (obtained from logist settings)

```
// Begin SLS Algorithm
Candidate A = Candidate.SelectInitialSolution(random, vehicles, task_list); // create initial solution

// Optimization loop - repeat until timeout
boolean timeout_reached = false;
while(!timeout_reached)  {

    Candidate A_old = A;  // record old solution
    List<Candidate> N = A_old.ChooseNeighbours(random); // generate neighbours
    A = LocalChoice(N, A_old); // Get the solution for the next iteration

    // Check timeout condition
    if( System.currentTimeMillis() - time_start > timeout_plan ) {
        timeout_reached = true;
    }
}

// End SLS Algorithm

// Build plans for vehicles from the found solution
List<Plan> plan = PlanFromSolution(A);
```

1

2

3

# Select initial solution

1. Get the vehicle with largest capacity
2. Assign all tasks to largest vehicle as two sequential PD-Actions

```
int num_vehicles = vehicles.size();

// Initialise plans and tasks variables
List<List<PD_Action>> plans = new ArrayList<>();
List<List<Task>> taskLists = new ArrayList<>();
List<Task> allTasks = new ArrayList<>(tasks);


// initialize plans and task list
for (int i = 0; i < num_vehicles; i++) {
    plans.add(new ArrayList<>());
    taskLists.add(new ArrayList<>());
}
```

```
// Get the vehicle with the largest capacity
double vehicle_capacities[];
vehicle_capacities = new double[num_vehicles];
int largest_vehicle = MaxIndex(vehicle_capacities);
```

1

```
// Assign all the tasks to the largest vehicle
for (Task t : allTasks) {


    List<PD_Action> plan = plans.get(largest_vehicle);
    List<Task> tasks_vehicle = taskLists.get(largest_vehicle);

    // Add tasks to the end of current plan
    plan.add(new PD_Action(true, t));
    plan.add(new PD_Action(false, t));


    tasks_vehicle.add(t);
}
```

2

```
// calculate the cost of initial candidate solution
double initial_cost = 0.0;
    // accumulate the cost borne by each vehicle
for (int i = 0; i < vehicles.size(); i++) {
    initial_cost += ComputeCost(vehicles.get(i), plans.get(i));
}
```

# Choose neighbours: Changing vehicle

1. For each vehicle, obtain the first task of the vehicle
2. Transfer the task to a random vehicle with suitable capacity
3. Modify candidate variables accordingly

```
// 1 - GENERATE NEIGHBOURS BY CHANGING VEHICLES OF TASKS

// System.out.println("Generating neighbours by changing vehicles of task");

 List<Candidate> neighs = new ArrayList<>();   // List to hold generated neighbours


int num_vehicles = vehicles.size();

// Loop over all source vehicle ids
for (int vid_i = 0; vid_i < num_vehicles; vid_i++) {

   List<Task> vehicle_tasks = taskLists.get(vid_i);  // Get tasks of the vehicle

   // Pass if the vehicle is empty
   if (vehicle_tasks.size()==0) {
    continue;
   }

   // Get the first task of the vehicle
   int task_id = 0;
   double task_weight = vehicle_tasks.get(task_id).weight;    // Get task weight

   // Randomly choose another suitable vehicle
   int vid_j = random.nextInt(num_vehicles);
   // Loop until finding a suitable vehicle
   while (vid_i == vid_j || vehicles.get(vid_j).capacity() < task_weight) {
    vid_j = random.nextInt(num_vehicles);
   }

   // Add a change of t from vehicle i to vehicle j
   neighs.add(ChangingVehicle(random, task_id, vid_i, vid_j));
```

1

2

3

# Choose neighbours: Changing vehicle (fnc.)

- Updating taskLists:
  1. Create copies of old lists
  2. Remove the task from the taskList of source vehicle
  3. Place the task in the taskList of target vehicle
- Updating plans
  1. Create copies of old plans
  2. Remove the pickup and delivery actions of the task from source vehicle plan
  3. Insert the pickup and delivery actions of the task to the target vehicle plan
- Updating cost
  - Compute the difference of costs of individual vehicles' plans from the old cost
  - No need to recompute everything

```
// Get source (i) and target (j) vehicles
Vehicle v_i = vehicles.get(vid_i);
Vehicle v_j = vehicles.get(vid_j);

// 1 - Update task lists

// Compute old task lists for vehicles
List<Task> i_tasks_old = taskLists.get(vid_i);
List<Task> j_tasks_old = taskLists.get(vid_j);

// Get the task to change
Task t = i_tasks_old.get(task_id);

// Create new task list for i by removing the task to change
List<Task> i_tasks_new = new ArrayList<>(i_tasks_old);
i_tasks_new.remove(task_id);   // remove the task

// Create new task list for j by adding the task to change
List<Task> j_tasks_new = new ArrayList<>(j_tasks_old);
j_tasks_new.add(t);   // insert the task to task list


// Update the task lists
List<List<Task>> updated_taskLists = new ArrayList<>(taskLists);
updated_taskLists.set(vid_i, i_tasks_new);
updated_taskLists.set(vid_j, j_tasks_new);
```

1, 2

3

# Choose neighbours: Changing vehicle (fnc.)

- Updating taskLists:
  1. Create copies of old lists
  2. Remove the task from the taskList of source vehicle
  3. Place the task in the taskList of target vehicle
- Updating plans
  1. Create copies of old plans
  2. Remove the pickup and delivery actions of the task from source vehicle plan
  3. Insert the pickup and delivery actions of the task to the target vehicle plan
- Updating cost
  - Compute the difference of costs of individual vehicles' plans from the old cost
  - No need to recompute everything

```
// 2 - Update plans

// Compute old plans for vehicles
List<PD_Action> i_plan_old = plans.get(vid_i);
List<PD_Action> j_plan_old = plans.get(vid_j);


// Create a new plan for i by removing the task to change
List<PD_Action> i_plan_new = new ArrayList<>(i_plan_old);
// remove actions associated with the task
// Loop over all actions in the plan
    for (int act_ind = 0; act_ind < i_plan_new.size(); ) {
        PD_Action act = i_plan_new.get(act_ind);
    // remove pickup/delivery if it is associated with task t
    if (act.task == t) {
        i_plan_new.remove(act_ind);
    }
    else {    // care not to update the index if a task is removed
        act_ind++;
    }
}

// Create a new plan for j by adding the task to change
List<PD_Action> j_plan_new = new ArrayList<>(j_plan_old);

// insert pickup/delivery actions associated with the task to the beginning of the plan
j_plan_new.add(0, new PD_Action(false,t));
j_plan_new.add(0, new PD_Action(true, t));
// note that we don't need to check the weight since the vehicle is free initially (assuming capacity is sufficient)

// Insert the new plans of source and target vehicles to the plans of the generated candidate
List<List<PD_Action>> updated_plans = new ArrayList<>(plans);
updated_plans.set(vid_i, i_plan_new);
updated_plans.set(vid_j, j_plan_new);
```

1, 2

3

# Choose neighbours: Changing vehicle (fnc.)

- Updating taskLists:
  1. Create copies of old lists
  2. Remove the task from the taskList of source vehicle
  3. Place the task in the taskList of target vehicle
- Updating plans
  1. Create copies of old plans
  2. Remove the pickup and delivery actions of the task from source vehicle plan
  3. Insert the pickup and delivery actions of the task to the target vehicle plan
- Updating cost
  - Compute the difference of costs of individual vehicles' plans from the old cost
  - No need to recompute everything

```
// 3 - Update costs

// Compute old costs for both vehicles
double i_cost_old = ComputeCost(v_i, i_plan_old);
double j_cost_old = ComputeCost(v_j, j_plan_old);

// Compute cost of the new solution
double i_cost_new = ComputeCost(v_i, i_plan_new);
double j_cost_new = ComputeCost(v_j, j_plan_new);
double updated_cost = this.cost - i_cost_old + i_cost_new - j_cost_old + j_cost_new;
```

# Choose neighbours: Changing task order

- Choose a random task from each vehicle
- Randomly modify candidate variables to create a new ordering for the chosen task

```
// Loop over all source vehicle ids
for (int vid_i = 0; vid_i < num_vehicles; vid_i++) {

  List<Task> vehicle_tasks = taskLists.get(vid_i);  // Get tasks of the vehicle

  // Pass if the vehicle has less than two tasks
  if (vehicle_tasks.size()<2) {
   continue;
  }

  // Get a task from the vehicle randomly
  int task_id = random.nextInt(vehicle_tasks.size());

  // Change the position of pickup and delivery actions of the task and add to neighbours
  neighs.add(ChangingTaskOrder(random, task_id, vid_i));
```

# Choose neighbours: Changing task order (function)

1. Remove pickup and delivery actions associated from the plan
2. Place pickup action in a suitable place
   - Loop until we find a place that satisfies weight constraints
3. Place delivery action in a suitable place
   - Ensure that it is placed after the pickup

- Cost update as it was before

```
// 1 - Update Plans

List<PD_Action> i_plan_old = plans.get(vid_i);   // retrieve old plan of the vehicle
List<PD_Action> i_plan_new = new ArrayList<>(i_plan_old); // create template for new plan

// remove pickup/delivery actions associated with the task for new plan
// Loop over all actions in the plan
for (int act_ind = 0; act_ind < i_plan_new.size(); ) {
    PD_Action act = i_plan_new.get(act_ind);

    // remove action if it is associated with task t
    if (act.task == t) {
        i_plan_new.remove(act_ind);
    }
    else {    // care not to update the index if a task is removed
        act_ind++;
    }
}
```

1

```
// insert the pickup action to a suitable place
int vehicle_capacity = v_i.capacity();
int pickup_location = 0;
List<PD_Action> candidate_plan_pickup = new ArrayList<>(i_plan_new);

// pick a random pickup location
boolean done = false;
while (done==false) {

    pickup_location = random.nextInt(i_plan_new.size());

    // add pickup action to candidate plan
    candidate_plan_pickup = new ArrayList<>(i_plan_new);
    candidate_plan_pickup.add(pickup_location, new PD_Action(true, t));

    // check if candidate plan satisfies weight condition
    if(SatisfiesWeightConstraints(candidate_plan_pickup, vehicle_capacity)) {
        done = true; // found pickup location if satisfies
    }
}
```

2

```
// insert the delivery action to a suitable place
List<PD_Action> candidate_plan_delivery = new ArrayList<>(candidate_plan_pickup);
// pick a random delivery location
done = false;
while (done==false) {

    // do not allow placing before pickup
    int delivery_location_offset = random.nextInt(i_plan_new.size()-pickup_location);
    int delivery_location = pickup_location + 1 + delivery_location_offset;

    // add delivery action to candidate plan
    candidate_plan_delivery = new ArrayList<>(candidate_plan_pickup);
    candidate_plan_delivery.add(delivery_location, new PD_Action(false, t));

    // check if candidate plan satisfies weight condition
    if(SatisfiesWeightConstraints(candidate_plan_delivery, vehicle_capacity)) {
        done = true; // found delivery location if satisfies
    }
}
```

3

# Local choice

1. With probability p, return the old solution.
2. With probability 1-p, return the best among the given set of neighbours.

```
if (random.nextFloat() < p) { // Return A with probability p
    return A;
}                                                                    1
else {    // Return the best neightbour with probability 1-p                2
    int best_cost_index = 0; // index of the neighbour with best cost until now
    double best_cost = N.get(best_cost_index).cost; // cost of the neighbour with best cost until now

    for (int n_ind = 1; n_ind < N.size(); n_ind++ ) {

        // check if current alternative has lower cost than the current best
        if( N.get(n_ind).cost < best_cost ) {
            // if so, update the best solution
            best_cost_index = n_ind;
            best_cost = N.get(best_cost_index).cost;
        }
    }
    // return the best solution
    return N.get(best_cost_index);
}
```

# Generating plan from a candidate solution

- For each vehicle:
  1. Traverse throughout the actions in their plan
  2. Add movement primitives to plan
  3. Add pickup or delivery primitives depending on action type to plan

```
// Build plan for each vehicle
for (int vehicle_ind = 0; vehicle_ind < A.vehicles.size(); vehicle_ind++) {

    Vehicle v = A.vehicles.get(vehicle_ind);

    // get constructed plan of the vehicle
    List<PD_Action> plan = A.plans.get(vehicle_ind);

    // follow vehicle cities to construct plan
    City current_city = v.getCurrentCity();
    Plan v_plan = new Plan(current_city);                    1

    // Append required primitive actions for each pickup/delivery action
    for (PD_Action act : plan) {

        City next_city;
        if(act.is_pickup) {
            next_city = act.task.pickupCity;
        }
        else {
            next_city = act.task.deliveryCity;
        }

        // Append move actions
        for(City move_city : current_city.pathTo(next_city)) {      2
            v_plan.appendMove(move_city);
        }
        // Append pickup-delivery actions
        if (act.is_pickup) {
         v_plan.appendPickup(act.task);
        } else {                                                    3
         v_plan.appendDelivery(act.task);
        }
        current_city = next_city;
    }

    // add plan to the list of plans
    plan_list.add(v_plan);
}
```