Logistics: documentation

The problem seems too hard to solve optimally for each input. Because our solution has to work for every input, we must use an approximation algorithm, otherwise we would risk non-polynomial running time for some inputs, which is clearly unacceptable.

Model of the problem

The problem consists of *M* cities, each city *m* contains *N_m* depos. Each city has a single depo designated as the airport. There are *P* parcels, each parcel has a source depo and a target depo, possibly in different cities. Parcels can be moved by means of trucks, which can travel between any pair of depos in a single city, and using airplanes, that can move between any pair of airports.

There are three kinds of actions (*go*, *load*, *unload*) for each type of vehicle (truck or airplane). Each *load* action loads a parcel from a depo to a vehicle located at the depo, and *unload* moves the parcel from the vehicle to its current depo. The *go* action transfers the vehicle to the target depo (which must be a valid target for the vehicle: depo in the same city for a truck, or any airport for an airplane), together with all parcels that are currently loaded into the vehicle. At no time can a vehicle contain more parcels than its capacity (4 for trucks, 30 for airplanes).

Algorithm

First, we observe that In the optimal solution, every parcel that must travel between cities is first carried by trucks to the source airport, then transported by airplanes to the target airport, and then again carried by trucks to its destination depo in the target city. Conversely, a parcel that does not have to travel between cities is never loaded into an airplane and is only carried by trucks between depos in the same city.

Subproblems

This suggests the first simplification: we can split the problem into an airplane routing subproblem and truck routing subproblems for every city separately. The airplane routing problem can assume that all parcels are already present at the airports, and a truck routing problem must carry all outgoing parcels to the airport before it can distribute the incoming parcels from the airport to the target depos. This reduces the set of possible plans for the trucks at each city, but we conjecture that the optimality gap resulting from this restriction is small for non-pathological inputs.

The airplane routing problem and the truck routing problems are similar. In the following, we will first describe the airplane problem and its solution, and only then generalize to the truck problem. The subproblem is still too hard to be solved exactly, so we decompose the solution into three approximation steps.

Step 1: planning edges

We simplify the problem by assuming that loading and unloading parcels is free. This is reasonable because the price to load and unload a fully loaded airplane (30*25 = 750 euro) is smaller than the price to fly the airplane (1000 euro). Therefore, it makes sense to first try to optimize the number of flights, and only later try to minimize the number of loading-unloading pairs.

As a second simplification, we plan only individual flights from an airport to another airport, without joining them into connected journeys. This is equivalent to assuming that we have an infinite supply or airplanes at every airport. It means that we will have to schedule some extra flights later to connect them, but in practice not many extra flights are needed.

Model of the problem

Define a square matrix *P*, where *P(i, j)* is the number of parcels going from *i* to *j*. An edge *e* travels from source airport *e_i* to target airport *e_j*, carrying *cargo(e)_k* parcels destined to airport *k*. When edge *e* is executed, it subtracts the vector *cargo(e)* from row *e_i* of matrix *P(i, j)* and adds it to row *e_j*.

To respect the limited capacity of an airplane, *cargo(e)* must be a vector of non-negative integers with sum at most 30. We call this difference the *free capacity* of the edge. We also cannot load more parcels than available at the source airport, so we must have *cargo(e)_k <= P(e_i, k)* for every *k*. This ensures that every entry of *P(i, j)* is always non-negative. Our goal is to find a valid sequence of edges *e* that makes *P(i, j)* diagonal, so that every parcel is moved to its destination.

In fact, we do not find a sequence of edges *e*, but only a partial ordering, to avoid commiting to a particular order earlier than is necessary. The partial ordering is represented as a set of constraints *C*, where *(e, f) in C* if edge *e* must be planned before edge *f*, because *f* needs to load parcels unloaded by *e*. (We call *C* the *is-before* relation in code.)

Approximate solution

However, even the simplified problem still seems to be too hard to solve exactly, so we must resort to an approximation. For each pair *(i, j)* of airports, we first add enough edges carrying 30 parcels from *i* to *j* to bring *P(i, j)* below 30. Then, we try to find an *augmenting path* from *i* to *j* in the graph *E* of edges that we have already planned, respecting the constraints *C* and using only edges with free capacity at least *P(i, j)*. If such a path is found, we send the remaining parcels through it, otherwise we add an extra edge going from *i* to *j* that carries the parcels.

To find an augmenting path, we use a simple breadth-first search, checking that the order of edges in the partial paths does not violate constraints *C*. Note that this algorithm is not complete: once it decides to use an edge, it never backtracks, so it may fail to find a path even if one exists. (Of course, with no constraints in *C*, a path is always found if one exists.)

The order of *(i, j)* pairs is arbitrary, so we sort the pairs in descending order of *P(i, j) mod 30*. The rationale is that we schedule the smaller *P(i, j)* at the end to make it more likely that an augmenting path will be found and no new edge will have to be added. We take a modulo because we always add at least *P(i, j) div 30* edges carrying 30 parcels and only the remainder can be routed through augmenting paths.

Note that this algorithm uses at most *sum_i,j ceil(P(i, j) / 30)* edges. The optimal solution uses at least *sum_i,j floor(P(i, j) / 30)* edges, so the difference between the optimal and approximate solutions is at most *N^2*, where *N* is the number of cities.

Step 2: planning journeys

In the second step, we lift the assumption of infinite number of airplanes and assign the edges to airplanes, forming connected journeys (trails in the graph of edges that begin at starting positions of airplanes). The legs of these journeys follow the edges from step 1, but we may have to add extra *jumps* that do not carry any parcels and their sole purpose is to move the airplane to another airport. The output of the algorithm is a totally ordered sequence of legs; the sequence of edges corresponding to these legs (ignoring the jumps) must

be consistent with the ordering imposed by constraints *C* from step 1. Our objective is to minimize the number of jumps.

Note that if we ignored the constraints *C*, the problem would be very similar to the problem of finding an Euler path in a directed graph. With a single airplane and a connected graph, we could find an optimal solution in polynomial time using depth-first search; the optimal solution jumps only from vertices with positive directed degree (where *directed degree = out degree - in degree*) to vertices with negative directed degree.

We use a simple greedy algorithm that generates the legs in chronological order without backtracking. We start with airplanes at their starting airports, and let them follow outgoing edges until all airplanes are stuck at airports with no outgoing edges. Then we use a heuristic to select an airplane to perform a jump, and fly with it to another airport, where it can continue following edges.

The airplanes prefer to follow edges that lead to vertices with high directed degree, attempting to minimize the probability of getting stuck. When selecting a jump, we pick the airplane at an airport with the smallest directed degree and jump to a vertex with at least one outgoing edge with maximum directed degree.

To ensure that the order of selected edges is consistent with *C*, we initially use only edges that are unconstrained. Additional edges *e* are added to the graph only when we have visited all edges *f* that come before *e* in *C*.

In the worst case, every visited edge is followed by a jump, so the number of legs produced by this algorithm is at most twice the number of edges produced by step 1.

Step 3: planning parcels

In the last step, we plan the loading and unloading of parcels into vehicles. Up to this point, we assumed that these actions are free, so we implicitly loaded all parcels at the start of the leg and unloaded them at the end of the leg. We also did not deal with individual parcels but only with their aggregate numbers.

Again, we use a simple greedy strategy to plan the loadings and unloadings. For each airport, we assign a list of parcels that are currently at this airport (not loaded into any airplane), and for each airplane, we maintain the list of all parcels that are currently loaded. Then we iterate over the legs from step 2 and for each leg, we emit the actions necessary to execute the work specified by the leg.

We must first ensure that the correct number of parcels for each destination is loaded before we fly the airplane, so first unload any extra parcels, adding them to the list of unloaded parcels at the current airport. Then we load the remaining parcels: first we try to take them from the airport and then, if none are directly available, we unload them from other airplanes at the same airport. When the correct amount of parcels is ready, we fly the airplane and then unload all parcels that are now in their destination.

If the set of edges from step 1 and the sequence of legs from step 2 is valid, then we can never get stuck and will always find an available parcel when it is needed.

The output of the third step is a list of actions *fly*, *load*, and *unload*.

Estimating the optimality gap

Our algorithm is only an approximation, it does not produce an optimal solution. However, we can estimate a lower bound on the cost of the optimal solution and thus find an upper bound of the *optimality gap* = *(approximate cost - optimal cost) / optimal cost*.

To compute the lower bound on the number of legs, we note that every solution must transfer *sum_j P(i, j)* parcels from airport *i*, so it must contain at least *ceil(sum_j P(i, j) / 30)* legs leaving airport *i*, or at least *sum_i ceil(sum_j P(i, j) / 30)* legs in total. Conversely, at least *sum_i P(i, j)* parcels need to be carried to every airport *j*,

so we need at least *sum_j ceil(sum_i P(i, j) / 30)* legs in total. We take maximum of these two values and multiply by 1000 to find the minimal cost of *fly* actions.

Trivially, every parcel must be loaded and unloaded at least once, so we multiply the number of parcels by 25 to obtain the minimal cost of *load* and *unload* actions.

Generalizing to the truck subproblem

Up to this point, we have dealt only with airplanes. However, a truck problem has only a single difference: we must carry all parcels from depos to the airport before we carry any parcel from the airport to a depo. This can be achieved with a few modifications to the approximation steps.

Planning edges

We mark edges with their stage, which is 0 if the edge carries parcels to the airport, 1 if it carries parcels from the airport and none if it carries other parcels. First we plan stage-0 edges and stage-1 edges and add constraints to ensure that every stage-0 edge comes before every stage-1 edge. Then we plan all other edges as before, possibly adding extra parcels to the edges from both stages.

A simple approach to generate the stage-0 edges is to simply add as many edges from every depo to the airport as necessary. However, this is exactly the worst case for step 2, which would have to generate a jump for every stage-0 edge. Therefore, we employ a simple greedy algorithm that tries to cover the vertices with the least amount of paths.

Of course, the same algorithm can also be used in the reverse direction to plan the stage-1 edges.

Planning journeys

We must ensure that all edges from stage 0 are planned before all edges from stage 1, so we allow the airplanes to use stage-1 edges only after all stage-0 edges have been visited. Otherwise, the algorithm is unmodified.

Planning parcels

The parcel planning algorithm does not have to be modified at all.

Of course, all three algorithms are implemented in a general way, so we use the same code for trucks and for airplanes. The algorithms use the terms *vertex* and *vehicle* instead of *airport/depo* and *airplane/truck* and can handle any number of stages.

Parallelization

The air subproblem and the truck subproblems are all independent, so they can be run in parallel.

Discussion

Our algorithm is a sequence of heuristics, so it does not find an optimal solution but runs in polynomial time. We hypothesise that the problem is NP-complete, so no optimal algorithm can run in polynomial time unless P=NP.

We did not perform any extensive experiments with the various heuristics, so it is likely that our algorithm can be easily improved. It may also be possible to improve performance by using heuristics specialized to particular kinds of subproblems.

Implementation

We implemented the algorithm in Rust. The program is split into several modules:

- `edge_plan`: the edge planning algorithm. Produces a vector of `Edge`-s and an object with the set of `Constraints`.
- `constraints`: the `Constraints` data structure, which maintains the partial ordering *C*. We store the complete set *C* as a bit array, so we have to propagate the transitive closure when a new pair is inserted into *C*, but we can test membership in *O(1)*.
- `journey_plan`: the journey planning algorithm. Takes the vector of `Edge`-s and `Constraints` and produces a vector of `Leg`-s, assigned to vehicles, in chronological order.
- `parcel_plan`: the parcel planning algorithm. Takes the vector of `Leg`-s and produces a vector of `Action`-s.
- `read`: reads the input file and prepares the subproblems.
- `main`: the entry point, solves the subproblems and collects the solutions into a single list of actions.

In total, the complete program (including some unit tests for `Constraints`) weights less than 1700 physical lines of code. We depend on a few crates (libraries) from crates.io: `array2d` (two-dimensional array data structure), `clap` (command-line parsing), `fnv` (fast hash function for small keys), `indicatif` (progress bars), `rayon` (parallelization).

Experimental evaluation

In this section, we will evaluate the performance of our algorithm. We are interested both in solution quality in terms of cost and in the running times. For each experiment, we report:

1. Cost of the solution
2. Cost per parcel
3. Optimality gap as computed by the program
4. Running time in seconds

The input files are generated by the provided `Generator.exe`. This tool creates an input with the given number of cities, given number of depos uniformly distributed into the cities, and a given number of parcels with sources and targets randomly sampled from the set of depos. We provide the input files used to generate the reported results in the source code repository. All outputs of our program were checked for correctness by the `Evaluator.exe` tool. It is puzzling that this tool is sometimes slower than our planner by an order of magnitude.

All runtime measurements were performed on a PC with AMD Athlon 64 X2 (2 CPU cores) with 4.28 GB of memory, running Linux kernel with version 5.0.0-37-generic. The program was compiled in release mode with `rustc` version 1.41.0-nightly.

Experiment 1: scaling the air problem size

In the first set of experiments, we generate problems with *n* cities and *n* depos, so each city has only a single depo and we solve just the air subproblem. We use *n/8* airplanes, and we test both *2*n^2* parcels (dense case) and *20*n* parcels (sparse case).

Dense

n	Cost	Cost/parcel	Gap	Time
4	8 625	270	92 %	0.01 s
8	26 725	209	145 %	0.02 s
16	103 675	202	188 %	0.04 s
32	410 900	201	208 %	0.11 s
64	1 715 025	209	240 %	0.61 s
128	6 901 225	211	252 %	15 s
256	27 479 000	211	250 %	760 s

Sparse

n	Cost	Cost/parcel	Gap	Time
4	10 750	134	97 %	0.01 s
8	30 150	188	162 %	0.01 s
16	75 800	237	224 %	0.03 s
32	213 600	334	342 %	0.08 s
64	463 250	362	380 %	0.14 s
128	1 157 550	452	502 %	0.37 s
256	2 489 000	486	541 %	1.6 s
512	5 345 075	522	586 %	9.6 s
1024	11 403 075	557	637 %	74 s

Experiment 2: scaling the truck problem size

We now fix the number of cities to 4, and generate *4*n* depos and *n* trucks, so each city has *n* depos and *n/4* trucks. Again, we examine both a dense case (*2*n^2* parcels) and a sparse case (*20*n* parcels).

Dense

n	Cost	Cost/parcel	Gap	Time

4	10 278	321	104 %	0.01 s
8	13 174	103	31 %	0.01 s
16	36 656	72	16 %	0.03 s
32	138 492	68	14 %	0.05 s
64	550 846	67	15 %	0.57 s
128	2 177 299	66	15 %	25 s
256	8 764 341	67	15 %	1300 s

Sparse

n	**Cost**	**Cost/parcel**	**Gap**	**Time**
4	14 081	176	118 %	0.01 s
8	16 455	103	48 %	0.01 s
16	27 759	87	34 %	0.02 s
32	48 339	76	21 %	0.02 s
64	97 591	76	25 %	0.05 s
128	186 132	73	20 %	0.09 s
256	363 844	71	19 %	0.59 s
512	729 716	71	20 %	2.3 s
1024	1 460 555	71	19 %	15 s

Experiment 3: scaling number of parcels

In the last experiment, we have 100 cities, 2000 depos, 200 trucks, 10 airplanes and *n* parcels.

n	**Cost**	**Cost/parcel**	**Gap**	**Time**
10^1	18 917	1892	78 %	0.03 s
10^2	156 901	1569	108 %	0.06 s
10^3	654 100	654	334 %	0.17 s
10^4	2 724 527	272	240 %	1.5 s
10^5	13 378 417	134	81 %	49 s
10^6	84 879 623	85	16 %	1400 s

Discussion

Our algorithm is able to solve a wide range of problems from `Generator.exe`. The optimality gap estimated by the program is not very satisfactory, but the lower bound on the cost of optimal solution is very crude, so our solutions may be better than the computed gaps lead us to believe. In particular, the estimates for optimal solution in Experiment 1 Dense, are probably overly optimistic.

Even though the algorithm is polynomial, the degree of the polynomial is very high. The running time is dominated by the need to maintain the C data structure in the edge planning step, which has size quadratic in the number of edges, and the number of edges is proportional to the number of parcels. On the other hand, other variables in the input (number of cities, depos, trucks or airplanes) have little effect on the performance.