Local Search

knimmaga and jpattiz
Screenname : classical

Local Search

We use local search with the following propositions:

- Reverse Run: Given a single truck, select a random sub path of its route and reverse it.
- Kill-a-truck: Select a truck that's visiting some customers and delete it from our assignment, randomly reassigning all stops it was responsible for.
- Shuffle: Select a random truck and shuffle the order it visited all customers
- Splice: Select a random sub path of a truck's route, and cut it out. Assign the entire sub path to another truck.

Distribution

- split the work up among several machines. We maintain a best result so far at the host machine, and repeatedly send this list out to other machines
- Machines maintain their best result so far and also all their valid solutions so far
- Repeatedly choose randomly from this pool, and apply a proposition
- Distributed simulated annealing would have worked better, but we wanted to try a different way and this works decently well

Randomization

When a machine gets its list of best tasks to search from, it may randomly instead select to choose one of them and permute it several times and search from there as its baseline.

This adds an additional layer of randomness / ability to explore new spaces in the search space

The actual proposition functions are also heavily randomized

This is extra important because we distribute the work and need to make sure the machines aren’t all doing the same thing

Getting unstuck

- System decides it is stuck if it hasn't improved in 20 seconds
- Starts using "extreme" propositions
  - Two opt swap across all machines
  - Try to destroy a semi randomly selected longest truck route and split it into a couple of truck routes
  - Pick a single truck and try every possible permutation of its customer orderings
- These are heavier duty operations, so it doesn't necessarily make sense to use them as general purpose propositions, but they help to get it unstuck if it has reached a potenti

Initial Solution Generation

- We use a greedy initial solution generation (just keep adding things to each truck until the truck is full)
- If this fails, we use the results from it with a separate score metric and just apply our normal local search
- Our metric for invalid solutions is some big number M + the summed amount of overload that each overloaded truck has
- This way, solutions that are closer to being valid will have a lower score, and eventually the system should produce a valid solution if one exists