Master Thesis

Using Column Generation for the Pickup and Delivery Problem with Disturbances

Author:
Dirk Koning

Supervisors:
dr. ir. J.M. van den Akker
dr. J.A. Hoogeveen

Universiteit Utrecht

Thesis number: ICA-0319791

June 2011

Abstract

This thesis shows how both the PDPTW (Pickup and Delivery Problem with Time Windows) and a variant where travel times are non-deterministic (Pickup and Delivery Problem with Disturbances), can be solved by an approximation algorithm that uses Linear Programming with column generation. In the pricing problem – which is solved by Simulated Annealing – an Evolutionary Algorithm helps to execute better performing sequences of mutations more often. Before making the Linear Program integer, some requests will temporarily be removed to create columns that helps to find that integer solution easier.

This strategy outperforms existing algorithms on some of the PDPTW benchmarks.

When the disturbances are taken into account during computation, the solution quality improves much compared to algorithms that assume deterministic travel times on problems that are non-deterministic in reality.

Table of Contents

1 Introduction

Almost every transport company has that question: in what order do we visit our customers? Load should be picked up at one customer and delivered to another one, both within a given time window.

If there are just a few customers this is easy to solve, but if there are, say 400 customers then this problem becomes much harder to solve. Now what is the optimal planning such that the number of vehicles and the total distance is minimal?

This report describes an algorithm that was used to solve this problem, in literature known as the Pickup and Delivery Problem with Time Windows. The research was first focused on finding techniques to improve existing algorithms and secondly on improving the model, because in real-life for instance the travel times are non-deterministic.

It is assumed that the reader has already knowledge of Linear Programming with Column Generation. A good article about this subject has been written by Barnhart et al. [1].

1.1 Problem definitions

1.1.1 Pickup and Delivery Problem with Time Windows

In the Pickup and Delivery Problem with Time Windows (PDPTW) there are n transportation requests that should be accomplished with (1) as little vehicles as possible and (2) as little distance as possible. Each request $i \in N$ has a size of load q_i that should be transported from an origin i^+ (the pickup) to a destination i^- (the delivery). All origins N^+ and all destinations N^- have time windows $[e_i, l_i]$ defined in which the vehicle should arrive. There is also a service time at each location since vehicles usually don't depart at the same time as they arrive.

The vehicle should start and end in the depot and it is not allowed to temporarily drop load there.

The locations are represented with a point in a two dimensional area and the travel time from one location to another is equal to the Euclidean distance those two points.

Furthermore, each vehicle has a maximum capacity of Q.

In literature there are some variants of this problem definition. Sometimes the time window denote the interval in which service should take place, instead of arriving only, which of course can easily be transformed. Furthermore, the objective function might be different, like for instance it might include minimizing idle time. Instead of minimizing the number of vehicles there might be a predefined number of vehicles. Savelsbergh and Sol [12] presented some overview.

1.1.2 Pickup and Delivery Problem with Disturbances

The Pickup and Delivery Problem with Disturbances (PDPD) is a new variant of the PDPTW where the travel times and service times are non-deterministic using a normal distribution. Moreover there are areas that simulate traffic jams, i.e. for journeys that cross those areas a stochastic delay will be added to the travel time. The delay will depend on the departure time, so during rush hour the delay is larger than it is at noon.

The time windows are no longer hard constraints, but there will be some costs added depending on what time the vehicle arrives. However, if a vehicle arrives too early (before e_i) it will have to wait, like it is in the PDPTW.

To simulate hard constraints one could set the costs if the vehicle arrives up to l_i equal to 0 and the costs after l_i equal to some large number. This number should not be infinity, because there is always a small probability that the vehicle arrives too late.

The details of the PDPD instances will be discussed later.

1.2 Literature review

A lot of research has been done on the Vehicle Routing Problem (VRP), much less on the PDPTW and hardly anything on routing problems with stochastic traveling times. As far as I know this is the first research on non-deterministic travel times for the PDPTW.

Nevertheless, there are some papers that deal about uncertainty in the load of each request. An important factor in these problems is how to handle overload, so what will be done if a vehicle arrives at a customer that has more load than the currently available capacity of that vehicle?

Dror et al. [2] introduced the traditional recourse strategy for the Vehicle Routing Problem with Stochastic Demands (VRPSD) where in case of overload the vehicle has to return to the depot to unload (or reload) after which it will continue the same route. Novoa et al. [3] found better solutions because there it is additionally allowed that other vehicles continue with the rest of the route.

In the VRPSD the random factor has a lot of influence. Therefore it would have been better to reschedule the remaining problem each time (or sometimes) when some more information has become available. There seems to be no research that uses rescheduling in advance, probably because this will make the computation much heavier. Moreover in real world changing the schedule is not always preferable because customers and drivers need to be informed that the appointments have been changed.

There are a couple of papers about the PDPTW. Bent and Van Hentenryck [4] use a two stage algorithm where in the first stage the objective is to minimize the number of vehicles and in the second stage to minimize the distance. They used a tabu search algorithm which gave good results, but soon after the report was published these results were improved by Røpke and Pisinger [5] who used (among others) Simulated Annealing.

Dumas et al. [6] were the first to use column generation for solving PDPTW. They proposed a branch-and-bound method where they solve the pricing problem to optimality. Due to this restriction of solving to optimality they are able to handle problems with up to 55 requests.

Sol [11] used a branch-and-price algorithm and solved larger PDPTW instances. Moreover it was also applied to a real-life situation, including driving hours restrictions etc. To solve the pricing problem it first uses a simple heuristic and after that a labeling algorithm to solve the pricing problem to optimality. This was possible because most routes consisted of only three or four requests.

This research also uses column generation, but to be able to solve problems with longer routes (e.g. more than 20 requests), the pricing problem is not solved to optimality but it uses an approximation algorithm, Simulated Annealing.

Compared to other researches this one uses some additional techniques that result in a more powerful strategy. Besides that it will also handle the PDPD variant where travel times are stochastic.

2 Approach

There are many ways to solve the pickup and delivery problems. This paper deals about an approach that uses Integer Linear Programming (ILP) with column generation, a technique that often performs well for these kind of problems.

This splits the problem into two parts. The master problem is to select those routes that result in the best overall solution and the subproblem is to find good routes.

In the master problem the objective will be to minimize the sum of the costs of the selected routes. In the PDPTW the costs of a route is defined as the sum of the total distance in that journey and a large constant *C*, to satisfy the objective for first minimizing the number of vehicles and secondly the total distance.
In the PDPD the costs function is a little extended, which is discussed later on.

The master problem can be defined as:

minimize:

$$\sum_{j \in J} c_j x_j$$

subject to:

$$\sum_{j \in J} a_{ij} x_j = 1 \qquad \forall i \in N$$

$$x_j \in \{0,1\} \qquad \forall j \in J$$

Here J is the set of all feasible routes; c_j holds the costs of route j; x_j are binary decision variables indicating whether or not route j should be selected; and a_{ij} is 1 if route j contains request i and 0 otherwise, so each route corresponds with one column in the ILP.

The constraint requires that every request will occur in one route of the solution. This constraint could also be replaced by one that requires that every request should occur in *at least* one route, which will be discussed in section 6.4.

Since the number of valid routes is huge, the problem will be solved by column generation, where only a small subset of J is used. In the pricing problem new columns will be found to improve the master problem.

The pricing problem will be discussed later on.

The computation of the master problem consists of four stages:

- the warming-up stage, a warming-up stage to create an initial set of columns;
- the LP stage, where linear programming with column generation is used to create more columns;
- the reduced LP stage where some requests are temporarily removed from the problem to make it a little easier to find an integer solution in the last stage:
- the ILP stage that will return an integer solution.

The following chapters deal about these stages. The warming-up stage will be discussed after the three other ones, because this one has a lot in common with the pricing problem in the LP stage.

Something that is of importance for the last two stages is that the warming-up stage will also find some reasonably good integer solution.

At first it is only about the PDPTW; after that we will discuss the changes for the PDPD.

3 The LP stage

In the LP stage an LP relaxation is being applied to the problem, so the selection variables no longer have to be binary but can be a fractional value. When the problem has been solved for the current set of columns the dual variables indicate which new columns will be attractive.

For each route the reduced costs can be computed as shown by Sol [11]. The reduced costs will be equal to the costs minus the sum of the dual variables that correspond with the requests that are handled in that route. Adding columns with negative reduced costs might improve the master problem and therefore the pricing problem is to find routes with the lowest reduced costs.

This pricing problem is solved by a local search algorithm that uses Simulated Annealing. As initial solution it will use the routes whose decision variable had a positive value in the LP. These routes will be mutated by the Simulated Annealing algorithm and if the best found route has negative reduced costs, it will be added to the solver.

After those new columns were created and added to the master problem, the LP will be solved again to update the dual variables.

A few milliseconds turned out to be sufficient to find new columns with negative reduced costs. If only a few (or none) of the initial columns returned a new column that has negative reduced costs, that available time will be increased.

Note that when adding a new column to the LP it could be that the LP already contained a route with the same requests, but in a different order. If the new column has fewer costs, the old column can be removed from the LP, since that one will never be selected in an optimal solution.

3.1 Simulated Annealing parameters

Kirkpatrick et al. [7] introduced the Simulated Annealing algorithm. Again and again the solution will be changed a little bit. If the new solution becomes better, it will always be accepted; if the solution becomes worse it will be accepted with a probability of $\exp((c-c') / T)$, where c is the old costs of the solution; c' is the new costs; and T is the temperature, a parameter of Simulated Annealing that will be multiplied with a constant α (a little smaller than 1) after Q new solutions has been accepted. Hence, the probability of accepting a solution that is worse becomes smaller each time, which moves the solution to a local optimum.

In this pricing problem there are many short runs for the Simulated Annealing and hence Q is set to 1, which means that T will be multiplied with α each time when a new solution has been accepted.

Each time when Simulated Annealing is used to mutate another column, the temperature will be reset again, since the solution of the pricing problem should move away from the initial solution and converge to a new local optimum.

Since the optimal values for α and the initial temperature T are unknown, these parameters will be adapted at run-time, based on the ratio that has been accepted – regardless the solution became better or worse. A target that appeared to work well is to accept ca. 2% of the mutations during the first third of the available time and ca. 0.1% during the last third.

Suppose 60 milliseconds are available for generating columns, then after 20 milliseconds will be checked how many solutions have been accepted. If this is less than 1.8% then next time the initial temperature will be doubled and if it is more than 2.2% then the initial temperature will be halved. Something similar will be done for α based on the acceptance ratio of the last 20 milliseconds. However, instead of doubling α (which would cause values more than 1) it will double or half the half-life of the temperature, that is ${}^{\alpha}\log \frac{1}{2}$. This comes down to computing the next value of α as resp. $\sqrt{\alpha}$ or α^2.

Lemma: halving ${}^{\alpha}\log \frac{1}{2}$ comes down to taking α^2, i.e.
$\frac{1}{2} * {}^{\alpha}\log \frac{1}{2} = {}^{\alpha'}\log \frac{1}{2}$ equals $\alpha' = \alpha^2$ for $0 < \alpha < 1$.

Proof:

$$\begin{aligned}
\tfrac{1}{2}\cdot{}^{\alpha}\log\tfrac{1}{2} &= {}^{\alpha'}\log\tfrac{1}{2}\\
\frac{1}{2}\cdot\frac{\log\frac{1}{2}}{\log\alpha} &= \frac{\log\frac{1}{2}}{\log\alpha'}\\
2\cdot\log\alpha &= \log\alpha'\\
\log\alpha^2 &= \log\alpha'\\
\alpha^2 &= \alpha'
\end{aligned}$$

Something similar can be derived to show that doubling the half-life comes down to taking $\sqrt{\alpha}$.

3.2 Mutating a route

There are two types of mutations: mutations that add and/or remove request(s) (category A) and mutations that only change the order of the stops of one route (category B). The latter will try many modifications of the route and only keep the best one, so in the PDPTW these can only improve the route.

Mutations from the category A are:

add, remove, replace

Mutations from the category B are:

2-opt, moveToBestPosition, permute

Each time one mutation from category A will run, then some mutations from category B might run and after that Simulated Annealing will decide whether or not the new route will be accepted. In what order the mutations will run, will be discussed more detailed in chapter 4.

A difference to most other usages of Simulated Annealing in vehicle routing problems is that in one iteration the route can change quite much. An iteration does not simply consist of for instance adding one request at a random position. Instead, in one iteration there are several mutations in a row where each mutation will try many options and only keeps the best one that was found. This moves the solution nearer to a local optimum before calling the Simulated Annealing procedure.

The reason for doing this is that it makes the algorithm more powerful, especially when the temperature *T* is low. The neighborhood will now contain much more good solutions which are similar to the current solution, e.g. having almost the same requests but in a different order. To find those solutions the cost might first increase (e.g. after adding a request) but after that it will likely decrease by running the category B mutations. It can be hard to cross those barriers for algorithms that will likely reject the solution after the first small modification.

This strategy of first moving to a local optimum before calling the Simulated Annealing procedure might not work for every problem, but for these routing problems it turned out to perform better.

3.2.1 Add

The 'add' mutation will try to add another request to the route. There are two types of the 'add' mutation: adding the new request at the best position or adding it at a pseudorandom position. Both types will only add the request at a position where the capacity constraint is not violated (and of course the delivery should be after the pickup).

Not all requests are interesting to be added to the current route. At first of course if the request has been added already, but also if the dual multiplier of that request is not positive, because adding such a request will never decrease the reduced costs.

Furthermore, some requests are more likely to improve the current route than others, especially the ones that are in the same direction as the current route (where the direction will be measured from the depot to a stop). One could use this as a bias for choosing new requests, which will result in a more effective search method even though it does not compare time windows and distances.

Another approach is the following one, which turned out to work even better.

As a preprocessing step the relation between requests is computed each time when the LP has been solved: the 200 columns with the lowest reduced costs are taken from the solver and then for each request it is counted in how many of these columns it occurs together with each other request. Now, when selecting a new request to add, it will more likely pick one that occur often with one of the existing requests of the route.
Since requests with a stronger relation to the route have a higher probability to be added, less time will be wasted to requests that probably will not improve that route.

Instead of trying just one request this mutation will try a few ones and it will finally add the one that results in the lowest reduced costs.

3.2.2 Remove

The 'remove' mutation will just remove one request from the route. It will try a few random requests and it will finally remove the one that results in the lowest reduced costs.

3.2.3 Replace

The 'replace' mutation will first remove one request and then add one or two requests, either at a pseudorandom position or at the best position. So in total there are four variants.

Again, when adding a request it will more likely pick one that is more related with an existing request of the route, based on the current best columns in the solver. And this mutation will also try a few combinations of requests.

3.2.4 2-opt

A part of the route will be flipped around. Out of all possible changes the best one will be executed.

Parameters: after an improvement of at least *a* percent 2-opt will be executed again. This can be repeated at most *b* times.

3.2.5 MoveToBestPosition

This will move one request (both pickup and delivery) to its best position. It will move the request that improve the route most.

Parameters: after an improvement of at least *a* percent MoveToBestPosition will be executed again. This can be repeated at most *b* times.

3.2.6 Permute

This will try all permutations for a sequential part of the route. At first it chooses randomly the part that should be permuted and then it will try all permutations and it will keep the best one. This will be repeated several times, depending on the number of stops of the route.

Parameters: try all permutations of a subsequence of *a* stops and repeat that *b* * *cardinalityOfRoute* times. If the length of the route is at most *c* stops it will just try all permutations of the whole route.

a varies from 4 to 6. *c* is 1 or 2 greater than *a*.

4 Choosing mutations

Recall from last chapter that there are mutations of category A (add, remove, replace) and mutations of category B (2-opt, moveToBestPosition, permute).

Each time when a mutation of category A has been executed, either three or no mutations of category B will be executed to find a local optimum. In this chapter that first mutation is called the MutationHead and the sequence of optimizations (category B mutations) the MutationTail. Together it will be a MutationIteration.

After each MutationIteration, the Simulated Annealing algorithm will decide whether the current route will be accepted or restored to its state before the MutationIteration was executed.

An example of a MutationIteration is:
- **replace** *one* request by *two* other ones which will be inserted at a *random* position
- **permute** a subsequence of *4* stops and repeat that *1.2* times the cardinality of the route
- **move both stops of a request to the best position** and repeat that at most *3* times while the costs decreased at least *0.1%*
- **permute** a subsequence of *6* stops and repeat that *0.5* times the cardinality of the route

Which mutations result in the fastest improvement, will of course depend a lot on the current position in the solution space. It is not known what the best mutations are at that moment, so selecting these is another search process.

Good mutations should be executed more often. On the other hand, there should also be a lot of variation, because that will make the neighborhood larger. There are too many possibilities to list all possible MutationTails. For these reasons, an Evolutionary Algorithm looks interesting. It does not matter that it does not find the ultimate MutationIteration, because such a thing does not exist. Instead, different MutationIterations should be executed.

To summarize, to search for new columns (routes) that hopefully decrease the value of the master problem, several mutations will be executed on some route and Simulated Annealing will accept it or not. For deciding which mutations should be executed an Evolutionary Algorithm is used. It will mutate the MutationIterations.

Créput et al. [11] tried an Evolutionary Algorithm to solve the entire PDPTW, but that did not perform well. However, for this 'problem' of choosing the mutations it did, as the alternative is to run the mutations randomly.

4.1 Evolutionary Algorithm

In general an Evolutionary Algorithm works as follows. It holds a number of (candidate) solutions, known as the population. The algorithm has some selection method which finds some of the best solutions. These solutions will generate offspring, usually by combining two solutions and/or mutating a copy.

Some of the worse solutions will be replaced by these generated solutions and this process will be repeated until some stop criterion is met. It is not required that the population holds the same size, but in most implementations it will.

The idea behind the Evolutionary Algorithm is that bad solutions will be eliminated and good solutions will create more good variants which are potentially even better, so the entire population will slightly improve.

The following sections describe how the MutationIterations can be adapted by an Evolutionary Algorithm. First section is about creating an initial population, next about selecting the best MutationIterations, then a section about creating offspring and the last one is about the termination condition.

Something that differs from most applications of the Evolutionary Algorithm, is that here it is not a continuous process that solves a problem, but instead the population is being used to solve another problem and the Evolutionary Algorithm is interrupted each time after a MutationIteration was executed.

4.1.1 Initialization

The population consists of 500 MutationIterations. Initially all mutations of category A have an equal number of instances in the population. For constructing the MutationTails the mutations of category B will pseudorandomly by picked such that the roughly guessed running time is equal for each mutation if all MutationIterations would run once. So a slow mutation will occur less often in the population than a fast one.

Only half of the population the MutationTail will consist of three mutations; the other half of the population will do no mutations after the MutationHead. It turned out that after running a while, the majority of the population will get a MutationTail, which means that running category B mutations after a category A mutation is a good idea, even though it takes more time in each iteration.

4.1.2 Selection

Each time eight different MutationIterations are pseudorandomly picked from the population and these are all executed four times. Note that executing a MutationIteration will change some route for the pricing problem and that after executing one the LP stage might set a new route to be changed.

Both the obtained improvement and the execution time of each MutationIteration will be logged. If the solution became worse, the improvement will be counted as zero, since this step can simply be undone by Simulated Annealing.

When all eight MutationIterations has been executed four times those are ranked based on the average improvement per second. In the next generation the worst four MutationIterations will be replaced by mutations of the best four MutationIterations.

One exception is that for every MutationHead there should always remain at least one instance in the population and at most 250 instances of that MutationHead. This is to avoid that mutations die out and to keep some variance in the population. For instance, it could be that the 'add' mutations performs best, but if the 'remove' mutation is never called, the search process will soon get stuck in a local optimum.

4.1.3 Reproduction

When mutating MutationIterations, either two MutationIterations will be merged or one category B mutation will be replaced in the MutationTail.

Merging two MutationIterations will only happen if these have the same MutationHead and both MutationTails are not empty. In that case for instance the last two mutations from one MutationTail will be replaced by two mutations of the other one.

When a category B mutation is being replaced it will more likely be replaced by a fast mutation. The probability is distributed such that when a mutation is on average twice as fast as another one, the probability to be selected is also twice as much, regardless the expected improvement of that single mutation.

4.1.4 Termination

The stop criterion is simple: when the program stops executing MutationIterations, the Evolutionary Algorithm will not be continued.

5 The reduced LP stage

After a third of the total computation time the problem will be reduced a little bit by temporarily setting the decision variable of a route to 1. After a set of new columns are created the decision variable can have any variable again and the next decision variable will be fixed to 1.

Section 5.1 describes which decision variables will be fixed and section 5.2 describes how new columns will be generated.

There are two reasons why fixing variables improves the search strategy. Firstly, it reduces the remaining problem so making it easier to solve. Secondly, it helps to make the solution a little more integer since at least one column is no longer fractional. This enforces the other fractional columns to be restructured.

Temporarily reducing the problem to find another set of columns is an idea that was also applied by Diepen [8]. It will be explained in section 6.2, since the approach of this chapter differs too much from the approach of Diepen.

5.1 Selecting variables to fix

The decision variables that are most interesting to temporarily fix to 1, are the ones that are likely to occur in an integer solution. Therefore it will iterate over the highest decision variables, but before that it first iterates one time over all positive variables of an integer solution.

That integer solution will be created by the MIP solver based on the current set of columns. The time spent for searching an integer solution is limited to one minute. Because the number of columns in this stage is much smaller than in the ILP stage it is often possible to find a good solution within that time limit. In many cases this is not the optimal solution, but that is not required here. The columns that are being generated based on the columns from the solution, will be useful anyway.

If there was no solution found within that minute, the best solution from the warming-up stage will be used. Recall that the warming-up stage was executed before the LP stage, but it will be discussed in chapter 7.

To summarize, the variables that will be fixed one by one are at first all positive variables of one integer solution and after that some more decision variables from the LP to start with the one with the highest value.
The number of variables that are being fixed will depend on the running time; on average this is circa 100 columns.

5.2 Creating new columns

When a decision variable is temporarily fixed to 1, all columns containing some of the requests of the corresponding route will be removed from the LP solver, because an optimal solution will not contain duplicates. For each column (but limited to the best 1000 columns) that has temporarily been removed, there is an attempt to reuse it: each invalid request and possibly some requests of which the corresponding dual variable is almost zero or less, will be removed from that route. After that the route is tried to be improved by adding some new requests and finally the sequence will be reoptimized. If the new route has negative reduced costs will be added to the solver.

After those removed columns were reused the problem is solved again and more columns will be generated as described in the LP stage.

In the next iteration the original problem will be restored, the removed columns will be added to the solver again and a new decision variable is picked to be set to 1.

This 'reduced LP stage' contributed much to find a better ILP solution faster.

6 The ILP stage

It turned out to be hard to make the solution binary when the number of columns grows. Most instances were not solved within an hour by the MIP solver, so some other strategies where required. One strategy is the one described in previous chapter.

The first attempt was a branch-and-price algorithm which did not succeed since the branching tree became too large; next a heuristic was used to make the solution binary, but that one only worked well for small instances; finally it turned out that the MIP solver was able to solve most problems if it has the right settings and if the right constraints were added to the problem, as discussed in section 6.3 and 6.4.

Although the first two strategies are not used anymore, these will quickly be explained below, section 6.1 and 6.2.

6.1 Branch-and-price

In the solution of the LP it is possible (or even likely) that some columns will be selected only partial, meaning that the corresponding x_j is between 0 and 1. Since the solution should be integer the first attempt was to branch as suggested by Ryan and Foster [9]. If the solution is not binary there will always be at least two requests that occur together in some of the partial selected column(s) and do not occur together in some other partial selected column(s).

The problem will be split into two subproblems, namely: either the two requests will be delivered by the same vehicle or the two requests will be delivered by two different vehicles. In both cases many columns can be removed from the problem.

In both subproblems new columns can be added if these match the branching criteria. When quite some columns have been added, the problem can be split again.

However, it turned out that branching this way took too much time, even if no new columns were created. Even if one would continue to a depth of 20 there is still a lot of flexibility to create valid routes and in many cases the solution will still not be binary. Yet there are over two million LP problems to solve if no branch is cut off.

The branch-and-price algorithm did not cut off any branch because no absolute lower bound known. The LP value can not be used as lower bound because new columns can be added in each branch. One could have tried some heuristic to estimate a lower bound. This will help a bit, but because the gap between the LP value and the best-found integer solution is often quite high, still too many branches cannot be cut off.

Therefore the next attempt was a heuristic to make the solution binary.

6.2 Heuristically rounding off

One heuristic could have been to round off some decision variables to 1 one by one, but a better heuristic is to do this more smoothly. The heuristic that was used, was adding constraints to first round off one variable out of three, i.e. a constraint $x_i + x_j + x_k \geq 1$ was added, where i, j and k are the indices of the columns that looks interesting to round off.

In each iteration a set of new columns was added to the problem and after that the constraints were narrowed, i.e. $x_i + x_j + x_k \geq 1$ was replaced by either $x_i + x_j \geq 1$, $x_j + x_k \geq 1$ or $x_i + x_k \geq 1$; and in the same way $x_l + x_m \geq 1$ was replaced by either $x_l \geq 1$ or $x_m \geq 1$. Each time the option that resulted in the lowest LP value was chosen.

After that, a new constraint was added the same way.

It does make sense which variables will be rounded off. When half of the routes has been rounded off to 1 then preferably these routes are all on the same half of the driving area so there is most freedom for the other half.

Therefore the heuristic started rounding off a route in some specific direction from the depot and change that direction after each iteration to realize that the routes that are rounded of are closely to each other.

The actual algorithm was slightly different because preferably variables with a higher value will more likely to be rounded off. Therefore it was computing some score that depended on both components. At the end it took the three routes with the highest score. At least one of these was required to be selected in the solution, i.e. a cut like $x_i + x_j + x_k \geq 1$ was added to the problem.

A route was only added to the cut if it did not have any request that was also contained in any other route of another cut. This was to avoid that the solution would have become infeasible when the cuts were tightened.

When almost every route has been rounded off it might be possible that there are no three routes that do not conflict with the existing cuts. In that case no new cut was added but the existing ones were tightened.

Each time when three cuts were added, one cut was removed. In the beginning the oldest cut was being removed; when there were almost no columns left, the one that increases the objective value most was being removed. Removing that cut will decrease the objective value most.

6.3 Creating additional columns

As the previous described heuristic sometimes gave quite bad solutions, the next step was to look for a way to let the MIP solver solve the ILP.

The first attempt was a heuristic that did work well for the gate assignment problem as shown by Diepen [8].

When one column gets selected for an ILP solution many other ones are not interesting anymore as these contain requests that are already covered by the first one. The heuristic therefore is to create second-choice columns, that is: each time when one new column has been generated one could take each request of the new column out of the problem one by one and solve the pricing problem again. The columns generated by this process could be set aside and added to the solver just before making the solution integer.

The gate assignment problem solved much faster when a few hundred thousand of these columns were added. However, for the pickup and delivery problem it did not help, at least not for the given instances of the Li and Lim's [10] benchmark. Attempts were made for adding up to 500,000 columns but due to those many columns even solving the LP took many seconds and the ILP could take hours.

Yet the solution value became better so as a trade-off the program will now add only a few thousand of those columns. Another difference to the approach of Diepen is that the columns will only be created just before making the problem integer, to get a better estimate of the dual values.

In this variant, one by one each request will temporarily be removed from the problem and so will all column containing that request. To avoid the LP solution to become worse, all columns that had zero reduced costs in the original LP solver will be collected. From these columns the request will be removed and that new column will be added to the LP.

Now the problem is resolved to get better dual values. Then for each column that has temporarily been removed, there is an attempt to reuse it by removing the request. From those smaller routes circa twenty of the best ones are taken to be added to a column pool. Moreover the algorithm will optimize some routes and try to add other requests. From these routes the best ones will be added to the column pool as well. Besides that, a few more columns will be generated as described in the LP stage.

When every request has been disabled once, all routes from the column pool will be added to the solver and the problem will be solved as ILP.

6.4 'Equals' vs 'at least'

The problem solves much faster when branch-and-bound tree is more limited. Something that helps a lot to make the solution less fractional is a constraint for the number of routes. If the sum of the variables of the current LP solution equals 30.4 for instance then you know that the best integer solution has at least 31 routes. Adding this constraint solves the ILP faster, but sometimes it still can take quite long, especially if the best known solution has 32 routes, since much time is spent to branches that will not find a solution with less than 32 routes.

First solving the problem for exactly 33 routes, then for 32 and finally for 31 ones solves faster than immediately solving it for 'at least 31 routes'. Moreover the MIP solver faster finds *some* solution which is better than returning no solution at all in case the time limit is being exceeded.

The constraint for the number of routes first will have the value derived from the best solution from the warming-up stage, as discussed in chapter 7. That number of routes will be decreased until no feasible solution can be found.

For the other constraints there is also the possibility to choose either whether every request should occur *exactly* once in the solution or *at least* once, which means that the = sign in the following constraint from the master problem can be replaced by a ≥ sign.

$$\sum_{j \in J} a_{ij} x_j = 1 \qquad \forall\, i \in N$$

During the LP process the sum of all variables containing the corresponding request will have to be *at least* one, mainly to avoid rounding errors in MIP solver that result in wrong dual values.

In the ILP stage one of the both can be chosen. If the sum should be *at least* one there are more valid solutions and since the ILP does not contain *all* valid routes, this solution (after removing duplicates) could be better. On the other hand, if the sum is required to be *equal* to one, the problem solves faster. The latter is chosen since (regarding the given time limit) it is then possible to add more columns in advance. If there are enough columns then no request has to be selected more than once.

As described in previous section some columns are added that simply has one request less than another column. This decreases the likeliness that the solution could have had duplicates.

7 The warming-up stage

The goal in the warming-up stage is to find an initial set of columns for the LP solver that already gives a quite good objective value.

At first an initial solution will be created. Then for several times (currently 40 times) the following actions are performed. These actions (except for the last one) will be explained in the subsequent sections.

- Try to decrease the number of routes.
- Mutate the solution for a few seconds using a Simulated Annealing algorithm.
- Move to a better local optimum using the RequestMover.
- Add all routes of the current solution to the LP solver.

7.1 Initial solution

The action to decrease the number of routes (as described in next section) turned out to work so well that an initial solution can simply consist of |N| routes where every route contains exactly one request. After merging the routes the solution is usually better than some greedy heuristic, since the latter usually ends with a few requests that increase the costs a lot.

7.2 Decreasing the number of routes

For each route in the current solution it will check if it is possible to move all requests of that route to any other route of the solution. Each request will be moved to the route where it increases the costs least. If not every request of the route can be reassigned to another one then the route will be restored.

This action will only be performed during the first few iterations or if minimal number of routes, according to the LP solver, is less than the number of routes in the current solution.

Note that in this stage the LP solver only contain the few columns that were added in the previous iterations of the warming-up stage. It does not yet give an absolute lower bound of the number of routes, but using the current one is a nice threshold that skips many calls to that time-consuming action at moments it probably does not find a better solution.

7.3 The Simulated Annealing algorithm

The algorithm for mutating a solution of the master problem is much similar to the one for the pricing problem.

One difference is that now the absolute costs of the entire solution are computed instead of the reduced costs of one route. The other difference is that it is no longer possible to simply add or remove requests since the solution should always be valid. Instead requests are moved from one route to another.

The mutations of category A are now: moving one request to another route, or swapping one to three requests from one route with one to three requests of another route.

The mutations of category B are the same. These will only operate on the two routes that have changed.

7.4 RequestMover

Since there is not much time spent to the Simulated Annealing stage, the solution can often be improved by just checking for each request if it better can be assigned to another route. It will search for the best position in each other route.

After the request was added the route might improve by applying some mutations of category B as described earlier. It would take too much time to optimize each route after trying to add the request. Therefore every request will temporarily be inserted at the best position of each other route, but only for the three routes where the costs increase the least after adding it, the stops will be optimized a little using some mutations of category B. The request will finally be moved to the best route.

8 PDPD

In the Pickup and Delivery Problem with Disturbances (PDPD) the travel times and service times are no longer a deterministic value, but these are a stochastic value that follow a normal distribution.

Both for each location (service time) and for each tuple of two locations (travel time) there is a function that has the departure time as input and returns the two parameters for the normal distribution.

The implementation would not change if that function returned another distribution, so with this implementation it is possible to simulate any real-life situation with respect to the travel times. The driving hours law is not yet taken into account, but that would not much change the way of solving either. It can always be solved by the local search algorithm that makes a small change to the route and reevaluates the route to compare the costs of the route.

In the deterministic case it was required to arrive on time at each location. In the stochastic situation this can never be guaranteed, but one would like some trade-off between minimizing the lateness and minimizing the distance. Therefore a penalty will be added, depending on the time of arrival. In real-life that penalty usually does not have to be paid, but since arriving too late will have negative impact on the customer relation, there will be some indirect costs.

The objective for the PDPD is to minimize the total costs, which is the sum of:

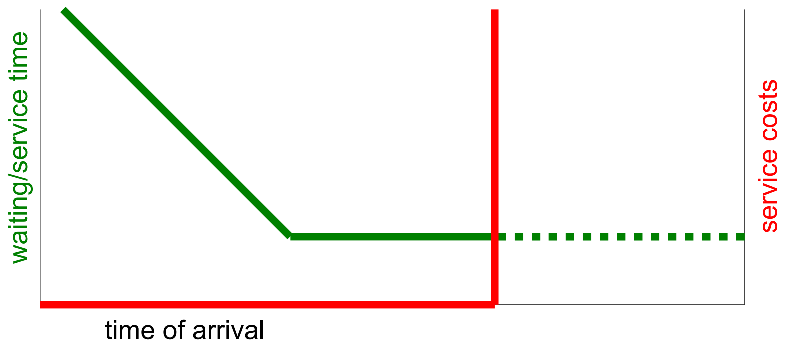
- the total distance
- the expected penalties for arriving too late
- a constant C for each vehicle that is being used

8.1 Creating instances

The instances for the PDPD will be based on the Li & Lim instances [10]. Next sections will describe the changes.

8.1.1 Time windows

In the Li & Lim instances all customers simply have one time window in which the vehicle should arrive. If the service costs and time would be drawn a graph, it will look like the picture below.



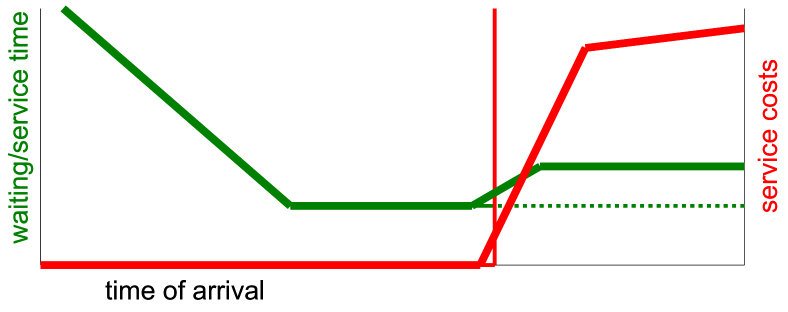
When the vehicle arrives before the end of the time window there are no service costs. However, when the vehicle arrives too late, the solution of the PDPTW becomes invalid. Therefore that can be modeled as service costs of infinity, which explains the vertical red line in the graph.

The green decreasing line shows that the vehicle has to wait until the beginning of the time window. During the whole time window the service time is equal. After the time window the service time does not matter anymore because these solutions will be rejected.

In the real world this is different: the time window constraint is softer and service times might depend a little on the time of arrival.

Let us first start with the end of the time window. The customer will prefer the vehicle to arrive on time, but arriving a few minutes too late is not as bad as arriving an hour too late. On the other hand, when the vehicle already is an hour too late, then waiting another hour is often not as bad as the first hour. In other words, it is often better to arrive two hours too late at one customer than arriving an hour too late at two customers. Of course there will be real-life situations where this is not the case, but for these PDPD instances it is.

The bold red line now shows a better reflection of the reality.



In most cases the service time will also increase when arriving too late, since for instance the right persons might not be immediately available.

There will also be customers for which the time window is less important. There the penalty for arriving to late will be less.

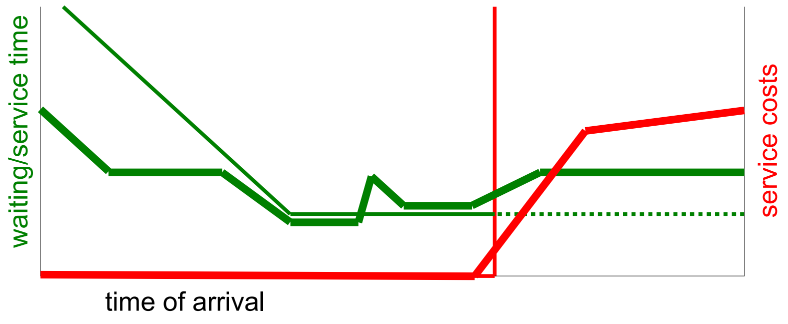
It could also be possible that arriving a few hours earlier is allowed, but with some additional waiting time. No costs will be added, to keep the computation easier, since it would be added, an optimal solution does not only depend on the order in which the customers are visited but also on how long the vehicle waits before going to the next customer. Of course the algorithm could be changed, but that is not in the scope of this survey.
To keep the comparison with the PDPTW fair, only a few customers will be created where arriving earlier is allowed.

Another variation is to introduce a break at noon. If the vehicle arrives at lunch time, it will have to wait for a little while.

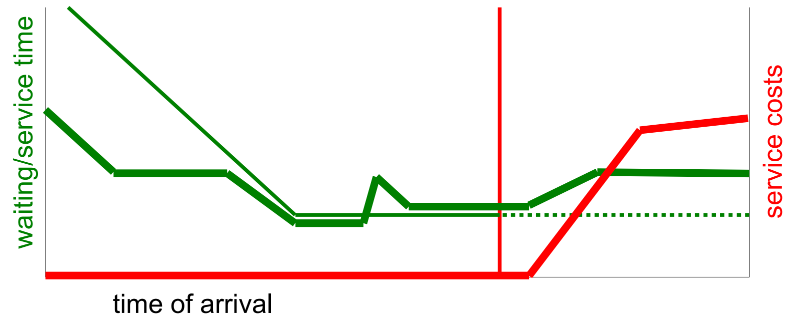
The last variation made is that it might be more attractive to arrive in the morning than the afternoon, or vice versa.

Of course, much more variations are possible, but for adapting the Li&Lim instances only these are performed. The implementation will not change if the service time distribution would have been more detailed.

Every customer will have one or more of these variations. All combined, the graph for a customer could look like the picture below. The peek in the middle indicate the lunch break.



If the real costs and service distributions are equal to the bold lines but the problem should be modeled as a PDPTW then the best estimate of the time window is equal to the thinner lines, although there can be some discussion about the exact value of the end of the time window. The end of the time window will depend on the costs of being too late and the estimated probability of being too late, based on the moment in time, so in the afternoon there will be a larger buffer in the PDPTW, i.e. larger time windows will be generated in the PDPD. Hence, the generated distribution of the previous picture will more look like the picture below, because the real arrival time of the PDPTW is usually later than the computed arrival time that did not take disturbances into account.



8.1.2 Traffic jams

In the real world roads are busy, especially on certain moments in certain areas. Nowadays very detailed information about traffic flows is available. For each moment in time one could compute an estimate of how long it takes to get from A to B.

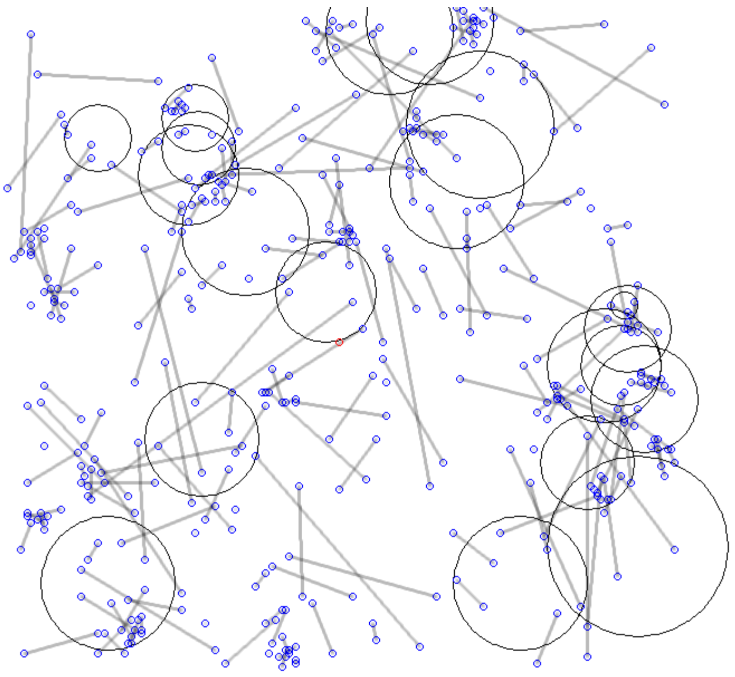
For each instance such a table will randomly be generated.

First twenty delay zones will be generated. These circular shaped areas are hard to get into during the morning rush hour and hard to get out during the afternoon rush hour. In the real world these areas are usually around the larger cities. Therefore a delay zone will be added near a cluster in the following way.

Firstly twelve triples of customers will randomly be picked from the instance. Then that triple will be used whose customers are most near each other, compared to the other eleven triples. The center of that triple will be the center of the delay zone. The radius of the delay zone will be equal to the distance from the center to any of the three customers.

Those values (the x, y position and the radius of the delay zone) will be rounded off to an integer value.

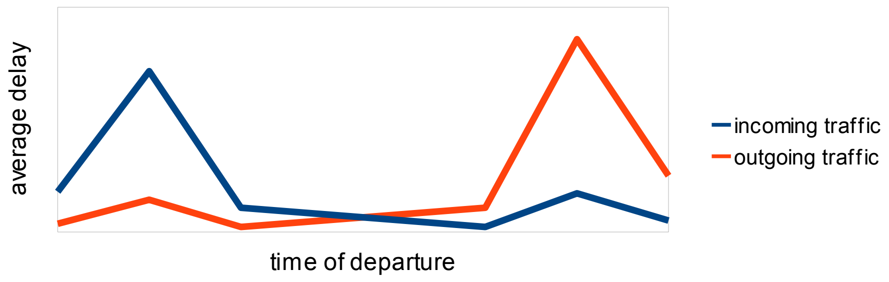
The picture of next page shows an example of an instance. The blue circles show the customers, where the pickup is connected to the delivery by a gray line. The red circle in the middle is the depot and the large black circles show the delay zones.



Each delay will get a weight, which is the maximum average of the delay when a vehicle crosses the delay zone. This weight will vary from 0.02 to 0.05 of a day length. To compute this weight a random value is taken from a uniform distribution, which is also the case for the forthcoming ranges mentioned in this section.

This maximum delay is only applied at two moments: one peak in the morning and one in the evening. Around those peaks the delay will linear increase / decrease. During the rest of the day the delay will be between 0.1 and 0.3 times the weight.

During the morning the delay zone will mainly (70 to 90%) affect the incoming traffic and the other 10 to 30% the outgoing traffic.



Note that a delay is normally distributed and therefore will not only consist of an average or expected delay but also a deviation. This deviation is between 10 and 30% of the average.

Finally a delay table will be computed that holds a mapping for each two customers: given a departure time from one customer it will return the delay (an average and deviation) if driving to the other.
To reduce the size of the delay table the values will be rounded off, so mappings that are equal can be combined.

8.2 Route computation

Computing the expected travel time of a journey becomes a little more complex with all those stochastic disturbances. Normally if two variables are independent normally distributed these can be combined to a new normal distribution. However, in this case the distributions are not independent, since departing 5 minutes later can cause arriving 10 minutes later, due to those time dependent functions for computing service and travel times.

Therefore the approach was chosen to run some simulations to get an estimate of the average costs of the route. The number of simulations will decide the quality of the estimate and also the performance decrease. The current implementation will do eight simulations as a trade-off between time and quality. Before adding a column to the master problem it will do 100 simulations to get a better estimate.

To compute the arrival times at the next stop one could simply iterate over the departure times of the current stop and compute the arrival times based on eight random simulations. The disadvantage of this approach is that selecting the best sequence will too much depend on luck instead of quality. The mutations will generate a lot of possible sequences and if computing the expected costs is a too random process it will simply select the sequence that had most luck during the eight simulations.

This could be avoided by running more simulations but that would decrease the performance too much. Therefore, instead of taking eight random values from the normal distribution the program simply takes those values where the cumulative probability is equal to 1/16, 3/16, 5/16 etc. In the simulation it takes a random permutation of these values, so when the first few simulations return a low value the later ones will return a high value.

Each time the travel time or the service time is being computed a new random permutation will be taken.

When a column will be added to the solver this approach is not used but 100 random simulations will be done to get a good estimate. When at the end a solution is being compared with one of the PDPTW 100 new simulations will be performed since the routes that were selected in the ILP solver will also have a little bias to the ones that had just luck during the first 100 simulations.

9 Performance aspects

When running an experiment it of course makes much sense how well the algorithm is implemented with regard to performance. This chapter describes most of the performance aspects that were taken into account at the detailed level, both for the PDPTW and PDPD, unless otherwise stated.

With all those improvements the program can find and evaluate (for the PDPTW) around six million valid routes per second on the PC on which the experiments ran. Here only the routes with a cardinality of 30 stops (excluding depot) were counted. Smaller routes will evaluate even faster.

When a route has changed, usually a lot of stops are still at the same position. Between two costs computations the part up to the first change does not have to be recomputed. Therefore the mutations of category B are implemented such that it will try to change the first part of the route as little as possible. The number of modifications is not changed, but the order in which these are evaluated are.

Another significant performance improvement is that all mutations will already take care about creating a valid sequence. Recall that each mutation will try many sequences and finally returns the best one. Now, instead of calling the cost function for any of the $n!$ possible sequences (where n is the number of stops in a route), the mutations will only call the cost function for sequences where every pickup is executed before its delivery and where the capacity constraint is not violated. This reduces the number of calls to the costs function a lot. Time windows are not taken into account by the mutation itself, because that would need arrival times of the whole route to be computed, which is almost as heavy as computing the costs. Note that the cost function returns a very high value if a time window constraint is violated and therefore such a route will never occur in the solution.

Next improvement is about the fact that often a limit for the costs exists. The category B mutations will only look for routes that improve the current best one. If the cumulative costs at the 20th stop of some route with 30 stops is already more than the costs of the best found sequence, then it makes no sense to compute the costs for the other 10 stops since this route cannot become better.

It is also possible to compute a lower bound beforehand. This lower bound will take the real costs up to the first change and takes only the distance for the other stops so it will not count the costs of being too late at the customers. It does count the dual multipliers. What it will also do is using that distance to compute a lower bound for the arrival at the depot.

Computing the lower bound improves the performance of the PDPD; for the PDPTW the time spent on computing the lower bound is about equal to the performance gain of computing costs.

9.1 More improvements

9.1.1 Caching

One improvement that is not implemented but looks interesting, is caching. After some mutations the route can be the same as before. The probability that this happens depends of course a lot on the size of the routes.

For the instances of 100 stops where the solution exist of 10 to 15 routes, caching could reduce the number of calls to the cost function with about 50% using a cache size of 10000.

However, for instances with 1000 stops where the solution exist of 30 to 40 routes this rate would be about 20%. It will also take some overhead to compare the routes for equality and because of the other performance improvements this might take more overhead than improvement.

9.1.2 Lazy simulation

For the PDPD the simulation is already quite lazy: during the mutations of category B (2-opt, MoveToBestPosition and Permute) it will only do the simulation if the lower bounds are lower than the costs of the previous route, but before and after a mutation of category A there will always be a simulation, even though it is likely that this route is not the best one.

Another issue is that during the costs computation there will always be a fixed number of simulations (eight simulations in the current implementation), even if after three simulations it is already clear that the current route is worse than the current best route.

These two issues could be solved by using a compare function for two routes. This compare function would only do that number of simulations until a certain confidence level is reached.

Of course, this would again take some overhead. The optimization of only computing the costs after the first change will now be a little harder because one part of the route might have had for instance five simulations already, the next part only three and the last part is completely new.

10 Results

10.1 PDPTW

The experiments ran on a Dell OptiPlex 755 machine with an Intel E6850 Core 2 Duo CPU of 3.0 GHz. The OS of the machine was Linux and the program was developed in Java.

The program used only one core, to keep the algorithm easier and to make it better comparable with other algorithms. In a real-life application where the computing time is limited you would of course use more than one CPU.

All instance got one hour running time. Based on some test runs the following time division was chosen: circa ten minutes were spent on the SA phase; ten minutes to the first part of the LP stage; during twenty minutes some routes were disabled and columns were generated based on the smaller problem; and during the remaining twenty minutes requests were disabled one by one to add new columns and finally solve the ILP. The time that was left after finding the ILP solution was not used.

To the instances of 100 customers only half an hour running time was given.

As MIP solver CPlex 11.0 was used.

Next page shows the results of the experiments.

The first column holds the name of the instance: the first two digits describe the size of the instance (e.g. 04 means 400 customers); the third digit is the type of the instance (usually type 1 has shorter routes than type 2); the fourth digit is the instance number (each type usually has 10 instances available); the characters after it describe the distribution of the customers: r = random, c = clustered, rc = a mix of random and clustered.

The second and third column hold the best-found solution by other people [10]. The last two columns show the solutions that our algorithm found in the one run of our experiment. Because the algorithm will make many random decisions it will not always find the same solution. For some instances a better result was found in a different run, but that one is not shown here.

If the solution found is better than the old best-found solution it is colored green; if it is worse it will be red; and it is gray if the distance is worse but the number of vehicles is less, or if the distance is better but the number of vehicles is worse. In those cases it does not matter much what the distance is, because the most important target for the PDPTW is to minimize the number of vehicles.

instance	best		found	
	vehicles	distance	vehicles	distance
0111c	10	828,94	10	828,94
0111r	19	1650,80	19	1650,80
0111rc	14	1708,80	14	1708,80
0112rc	12	1558,07	12	1558,07
0113rc	11	1258,74	11	1258,74
0114rc	10	1128,40	10	1128,40
0115rc	13	1637,62	13	1637,62
0116rc	11	1424,73	11	1424,73
0117rc	11	1230,15	11	1230,14
0118rc	10	1147,43	10	1147,43
0121rc	4	1406,94	4	1406,94
0122rc	3	1374,27	3	1374,27
0123rc	3	1089,07	3	1089,07
0124rc	3	818,66	3	818,66
0125rc	4	1302,20	4	1302,20
0126rc	3	1159,03	3	1159,03
0127rc	3	1062,05	3	1062,05
0128rc	3	852,76	3	852,76
0211c	20	2704,57	20	2704,57
0211r	20	4819,12	20	4819,12
0211rc	19	3606,06	19	3606,06
0212rc	15	3673,19	15	3671,02
0213rc	13	3161,75	13	3165,91
0214rc	10	2631,82	10	2631,82
0215rc	16	3715,81	16	3715,81
0216rc	17	3368,66	16	3572,16
0217rc	14	3668,39	14	3723,97
0218rc	13	3174,55	13	3146,70
0219rc	13	3226,72	13	3157,34
0221rc	6	3605,40	6	3595,18
0222rc	5	3327,18	5	3404,36
0223rc	4	2938,28	4	2984,91
0224rc	3	2887,97	4	2238,23
0225rc	5	2776,93	5	2776,93
0226rc	5	2707,96	5	2707,96
0227rc	4	3050,03	4	3044,40
0228rc	4	2399,95	4	2435,31
0229rc	4	2208,49	4	2214,43
0411c	40	7152,06	40	7152,06
0411r	40	10639,75	40	10639,75
0411rc	36	9127,15	36	9152,58
0412rc	31	8346,06	31	8346,58
0413rc	25	7307,09	25	7287,08
0414rc	19	5838,58	19	5809,50
0415rc	33	8773,75	32	8923,67
0416rc	31	8177,90	30	8423,70
0417rc	29	7992,08	28	8037,87
0418rc	27	7613,43	27	7563,09
0419rc	26	8013,48	26	7790,26
0421rc	12	7471,01	12	7659,68
0422rc	11	6303,36	11	6308,52
0423rc	9	5438,20	9	5584,58
0424rc	5	5322,43	7	4793,83
0425rc	11	6120,13	11	6220,40
0426rc	9	6479,56	10	5919,18
0427rc	8	6361,26	8	6663,20
0428rc	7	5928,93	8	5249,88
0429rc	7	5303,53	7	6068,59
1011c	100	42488,66	94	53287,54
1011r	100	56903,88	100	56744,91
1011rc	84	49315,30	84	48874,15
1012rc	73	45135,70	75	45356,63
1013rc	55	35475,72	58	35583,66
1014rc	40	27747,04	47	30717,70
1015rc	76	49816,18	77	49506,08
1016rc	69	44469,08	69	43922,25
1017rc	64	41413,16	69	42556,24

The reason why the instances of 400 customers of type 1 perform much better than the other ones is that the development was mainly focused on these instances. Some examples are:

- With all the parameters the number of columns in the solver were just fine to solve for those instances. Other instances might generate too much or too few columns for an optimal solution that can be solved in time.
- Especially in the reduced LP stage some variants were tried and the parameters were tweaked such that it worked best for the 400 instances.
- The mutations 2-opt and MoveToBestPosition have polynomial running time compared to the cardinality of the route. With twenty stops this is fine, but if the route consists of 50 stops this will relatively take too much time and some other techniques might be required.

For the instances of 1000 customers, CPlex often did not have enough time to find the optimal ILP solution, but if it did then the solution usually is better than the previous best-known solution. Simply increasing the available time for CPlex will not help much since the required running time seems to grow exponentially. It requires more research to tune the program such that these large problems can also be solved. Maybe less, more or different columns should be added to the solver. There might also be some strategy to split the problem into smaller subproblems.

10.2 PDPD

It turned out that converting the PDPTW instances has a large impact. The solutions found in the PDPTW became much less attractive because vehicles did often not arrive on time even not when the traffic delays were ignored. Moreover, when solving the problem as PDPD it turned out that much better solutions were possible, mainly because vehicles are allowed to arrive later which increases flexibility. Even though minimizing the number of vehicles is no longer the main objective, yet the number of routes was decreased with ca. 30%.

Of course this depends a lot on the parameters used at the conversion. Yet for every parameter a realistic estimate was made with even some bias to the PDPTW, e.g. adding a deviation to the travel time of 2% is not much.

In another attempt the penalty of arriving after the time window was increased. That made the solution of the PDPTW worse of course, but in the PDPD this was difference was less because the routes could be changed such that the likeliness of arriving too late was decreased.

11 Conclusion and future research

Introducing a deviation to the expected travel times showed that the solutions that seemed interesting in the deterministic case are not very robust. For the Li & Lim instances this difference is so large that even a simple local search algorithm for the stochastic case outperforms the best algorithm that assumes that travel times etc. are a deterministic value. Hence, for real-life situations it is worth considering to take the disturbances into account. It might depend on the situation how much the quality of the solution improves.

The disadvantage of the stochastic model is of course that it requires more computation time. If that is too limited it could also be interesting to investigate what the quality of the solution is if the column generation process uses the deterministic model but just before adding a column to the solver the stochastic model is being used to run several simulations. In that case the performance will almost be the same as the deterministic algorithm, but while selecting the routes in the master problem it will choose the routes that perform best in the non-deterministic model.
In this approach some correction for the dual values might be required to avoid that the pricing problem remains returning columns that perform well in the deterministic model but do not in the stochastic one.

Anyway, in a real-life situation it is an obvious step to make the program multi-threaded. The column generation approach is very well suited for cloud computing, because the local search algorithm has quite a high random factor so every processing unit will likely return different columns which can all be added to the master problem. To solve the master problem it is possible to configure CPlex such that multiple processor cores are being used.

In this research area there is a lot of work that can be done. As shown, the column generation approach works well for both the PDPTW and the PDPD. Still there are many tricks that will make the search process a little better. Smart selection of the mutations, run-time adaptation of parameters, temporarily removing some requests, caching the cumulative costs: these are just some of the improvements that were implemented in this research. And yet there will be much more ways to improve the algorithm.

Nevertheless, solving a real-life problem as good as possible does not only depend on finding the best algorithm for a model, but also on finding the best model for the problem.

12 References

1: Cynthia Barnhart, Ellis L. Johnson, George L. Nemhauser, Martin W. P. Savelsbergh, Pamela H. Vance, *Branch-and-Price: Column Generation for Solving Huge Integer Programs*, 1996. Operations Research 46, 316-329.

2: M. Dror, G. Laporte, P. Trudeau, *Vehicle routing with stochastic demands: Properties and solution frameworks*, 1989. Transportation Science 23, 166-176.

3: Clara Novoa, Rosemary Berger, Jeff Linderoth, Robert Storer, *A Set-Partitioning-Based Model for the Stochastic Vehicle Routing Problem*, 2006. Texas State University, 06T-008.

4: Russell Bent, Pascal Van Hentenryck, *A Two-Stage Hybrid Local Search for the Vehicle Routing Problem with Time Windows*, 2001. Transportation Science 38, 515-530.

5: S. Røpke, D. Pisinger, *An Adaptive Large Neighborhood Search Heuristic for the Pickup and Delivery Problem with Time Windows*, 2006. Transportation Science 40, 455-476.

6: Y. Dumas, J. Desrosiers, F. Soumis, *The pickup and delivery problem with time windows*, 1991. European Journal of Operational Research 54, 7-22.

7: Scott Kirkpatrick, C. Daniel Gelatt and Mario P. Vecchi, *Optimization by Simulated Annealing*, 1983. Science 220, 671-680.

8: D.M. Ryan, B.A. Foster, *An integer programming approach to scheduling*, 1981. Computer Scheduling of Public Transport, 269-280.

9: G. Diepen, J.M. van den Akker, J.A. Hoogeveen, J.W. Smeltink, *Using column generation for gate planning at Amsterdam Airport Schiphol*, 2007. Utrecht University, UU-CS-2007-018.

10: Li, Lim, *Li & Lim benchmark*, 2001, http://www.sintef.no/Projectweb/TOP/Problems/PDPTW/Li--Lim-benchmark

11: Marc Sol, *Column Generation Techniques for Pickup and Delivery Problems*, 1994. Eindhoven University of Technology.

12: M.W.P. Savelsbergh, M. Sol, *The General Pickup and Delivery Problem*, 1995. Transportation Science 29, 17-29.