0.1 Path Relinking

The solution polishing approach has a small search space which is rapidly exhausted. We identify path relinking as a controlled process to explore combinations of good solutions with the objective of developing a broader route search space (Glover et al. (2000)). The principle we implement maintains a pool of solutions from which an initial and guiding solution is selected. We then apply a set of neighbourhood operators to incrementally transform the initial solution into the guiding solution. We consider a solution space as the set of solutions that can be reached by iteratively applying defined neighbourhood operators to a solution. Because of the complex cross-route linking using cross-docks, an important feature of Split Pick-up and Delivery Vehicle Routing Problem with Cross-Docks (SPDVRP-CD) solutions is that the feasible time windows for each stop are often very tight. This means single alterations (i.e., adding or removing a vertex, replacing a vertex, or an arc swap) almost always result in an infeasible solution. However, they often generate a new route combining features of two solutions that is feasible in the context of a slightly different route pool.

We define distance between two solutions in a solution space as the differences in solution characteristics that are changed by applying neighbourhood operators. In the literature, the characteristics used are often vertices and edges. The neighbourhood operators are applied such that the resulting solution is closer in neighbourhood space to the guiding solution. In our implementation, we do not extend the path past the guiding solution. Path relinking contains some elements of a genetic algorithm (GA) where two solutions are selected and operators applied to develop new solutions sharing characteristics of the starting solutions. However, unlike a GA (Prins, 2004; Huang and Ting, 2011), the combinations are guided to explore the space between two feasible solutions. Because most intermediate solutions are not actually feasible, we do not evaluate feasibility at this stage. We identify new routes that combine elements of known successful solutions and these routes are then ranked with one of the selected metrics and introduced into the route pool for recombination with the pool of candidate routes to identify a new feasible solution.

In the following sections, we described the operators, measuring distance in solutions space, explain the rationale for using two different distance measures, and introduce new notation for defining the solution space and operators relevant to this space.

0.1.1 Notation

New notation is introduced for the algorithmic descriptions of the path relinking neighbourhood operators and processes. For stops at cross-docks ($x \in V_x$), it is necessary to distinguish between x being the origin or terminus of a route and x being a cross-dock visited in the middle of a route. To address this requirement, let V'_x be a set of copies

of cross-docks in V_{x}, where the vertices in V'_{x} are labelled with a prime version of the vertex number of the corresponding vertex in V_{x}. We then define an extended vertex set $V' = V'_{\mathrm{x}} \cup V$ to be the set of all vertices in G (defined in Chapter 0.2) combined with the cross-dock vertex copies in V'_{x}.

Let S^γ, $\gamma \in \mathbb{N}$ be the value of the decision variables for the γth solution of the MILP and $\bar{R}(\gamma)$ be the set of routes used in this solution. For the purpose of path relinking, when decoding the routes $r \in \bar{R}(\gamma)$, each stop $\sigma_r(k) = i$ for $k \in \{1, n_r\}$ is replaced by the corresponding $i' \in V'_{\mathrm{x}}$, resulting in all departure and arrival locations being in V'_{x} and all other stops, including stops mid-route at cross-docks in V_{x} being members of V. Additionally, the decoding maps the separate arrival and departure stops for intermediate cross-docks into a single stop for the purpose of path relinking with these consolidated stops replaced by the separate arrival and departure convention when mapped back to candidate routes.

Using the routes of solution $\bar{R}(u)$, decoded as described, and the expanded vertex set V', and a guiding solution $\bar{R}(g)$, the following sets and functions are defined:

$V_i^\gamma = \{(r,k) \mid r \in \bar{R}(u),\, \sigma_r(k) = i,\, k \in \{1,\ldots,n_r\}\}$, $\forall i \in V'$, a set of tuples of routes and stop numbers in S^u, servicing each $i \in V'$.

$\tilde{A}_{ij}^\gamma = \{(r,k) \mid r \in \bar{R}(u),\, k \in \{1,\ldots,n_r-1\},\, \sigma_r(k) = i,\, \sigma_r(k+1) = j\}$, a set of tuples of route and stop numbers composed of stops servicing i followed by j, for $i, j \in V'$.

$\mathrm{SEG}(r, k, S^u, S^g)$, $r \in \bar{R}(u)$, $k \in \{1,\ldots,n_r\}$, a function returning T if $\sigma_r(k)$ and $\sigma_r(k+1)$ are in different *segments* in S^u as defined in Section 0.1.2 and F otherwise.

$\bar{A}_{ij}^\gamma = \{(r,k) \mid r \in \bar{R}(u),\, k \in \{1,\ldots,n_r-1\},\, \sigma_r(k) = i,\, \sigma_r(k+1) = j,\, \mathrm{SEG}(r,k,S^u,S^g) = \mathrm{T}\}$, $\forall i, j \in V'$, a set of tuples of route and stop numbers composed of stops servicing i followed by j, where i and j are in different segments.

A_{ij}^γ is used in the pseudo-code to refer to either $\bar{A}_{ij}^\gamma$ or $\tilde{A}_{ij}^\gamma$, depending on which distance metric is currently in use in the algorithm, as described below.

For clarity in algorithmic descriptions in this section, we sometimes use the notation a_{ij} to refer to an $(r,k) \in A_{ij}^\gamma$, showing it represents the arc between $\sigma_r(k) = i$ and $\sigma_r(k+1) = j$. The assignment $(r,k) \leftarrow a_{ij}$ means r and k take the values of a a_{ij} selected from A_{ij}^γ.

$D_i(S^a, S^b)$, $a, b \in \mathbb{N}$, is a function returning the difference in instances of stops servicing vertex i between two solutions a and b, being $|V_i^a| - |V_i^b|$.

$D_{ij}(S_a, S_b)$, $a, b \in \mathbb{N}$, is a function returning the number of extra arcs in A_{ij}^a of S^a compared to A_{ij}^b of S^b, being $\max\{0, |A_{ij}^a| - |A_{ij}^b|\}$.

To ensure convergence to the guiding solution, two different measures of distance are required as discussed in Section 0.1.2. We define two metrics, $\delta^1(S^u)$ and $\delta^2(S^u)$, to measure the distance between the current solution S^u and the guiding solution S^g (which is always the same solution and omitted in this notation). The metrics are the sum of arc or segment differences between the current solution and the guiding solution. It is unnecessary to include vertex difference as each vertex difference necessarily results in 2 arc differences.

$$\delta(S^u) = \sum_{i \in V'} \sum_{j \in V'} (D_{ij}(S^u, S^g) + D_{ij}(S^g, S^u)).$$

When using δ^1, D_{ij} is calculated with A^γ_{ij} being $\tilde{A}^\gamma_{ij}$ and when using δ^2, D_{ij} is calculated with A^γ_{ij} being $\bar{A}^\gamma_{ij}$

We further define,

$I(S^u)$ The intrinsic cost of a solution, which is the transport cost summed over all of the routes, ignoring potential waiting time.

m A move consisting of deleting or inserting specified arcs and vertices in the current solution.

Finally, we introduce the concept of a degenerate route (x', x'), $x' \in V'_{\mathrm{x}}$. Such a route has no capacity to undertake a delivery or collection and does not contribute to the objective cost of a solution (if fixed costs are included for a route, these routes are defined to have a fixed cost of 0). Degenerate routes facilitate increasing the number of routes originating from a given cross-dock, which can be necessary for convergence to the guiding solution.

0.1.2 Solution Distance Measurements

The solution distance measures are based on encoding each solution as a sequence of routes, each route being a vector of stops, encoding a cross-dock used as the origin/terminus with a prime vertex number, (i.e. $i \in V'_{\mathrm{x}}$), and all other stops using the vertex number from V. All solutions have the route vectors sorted in lexicographical order to ensure routes appear in the same relative position (i.e., solutions with identical routes r_1, r_2 will always encode r_1 before r_2 regardless of the route index number). We primarily consider the arc differences between the current and guiding solution as the measure of distance in a solution space. This is appealing as it is computationally efficient and has precedence in the literature. However, due to our problem admitting multiple visits to a vertex, it is possible for two structurally different solutions to be composed of the same arcs. Thus, a solution distance of zero between two solutions does not guarantee the two solutions have the same set of routes.

Consider the two solutions

$$S^1 = (0,1,7,6,0)(0,3,4,2,1,5,0) \qquad S^2 = (0,3,4,2,1,7,6,0)(0,1,5,0),$$

with both solutions having the same arc set, but are clearly structurally different. As such a richer approach to measuring solution distance is developed called *segment difference*. We first define a *segment* as a subset of a route where a sequence of vertices exactly match in two solutions being compared. A *segment difference* occurs when extending the segment in one route causes it to no longer match any segment in the comparison soluton. The process is described below with pseudo-code provided in Algorithm 1.

1. Initialise BreakSet, the set of (r,k) tuples defining segment breaks, to the empty set.
2. While the solution is not fully segmented, do the following steps.
3. Select the next r from $\bar{R}(u)$ to segment.
4. Start a sequence, p with the first stop in r.
5. While all stops in r have not been allocated to a segment.
   - In line 6, create P, a set of all sequences in S^g identical to p that are not yet marked as included in a segment.
   - If the set P is not empty, that is unmarked segments in S^g match p, extend p by appending the next stop in r.
   - Otherwise, a segment break has been identified and lines 10-13 process the break. In line 11 a sequence $p' \in P$ from the guiding solution is matched to p and these vertices are marked in S^g as included in a segment. Then (r,k) is added to BreakSet.
   - In lines 13-14, if the route has not been fully segmented, a new sequence is created with $\sigma_r(k+1)$ as the initial element.
   - k is then incremented to point to the next stop in the sequence under consideration.
6. BreakSet is returned and is used in the function SEG (Equation 0.1.1) to create the A^{γ}_{ij} set for use in the secondary distance measure.

Algorithm 1 Segment Break

BreakSet $\leftarrow \emptyset$
for all $r \in \bar{R}(u)$ **do**
  $k = 1$
  $p \leftarrow (\sigma_r(k))$
  while $k \leq n_r$ **do**
    $P \leftarrow$ {sequences of stops $p' \in \bar{R}(g)$
      s.t. $p' = p$ and p' not marked as segmented}
    if $P \neq \emptyset$ **then**
      $p \leftarrow p$ with $\sigma_r(k+1)$ appended.
    else
      Select a $p' \in P$, mark as segmented in $\bar{R}(g)$
      BreakSet $\leftarrow$ BreakSet$\cup\{(r, k)\}$
      if $k < n_r$ **then**
        $p \leftarrow \sigma_r(k+1)$
    $k \leftarrow k + 1$
return BreakSet

An example of the segmentation process is shown in Figure 1, where the matched segments are shown in matching colours across the two solutions. The process is run from top to bottom of the initial route set and ties between equal segments in the guiding solution are broken selecting the segment towards the top of the graphic.

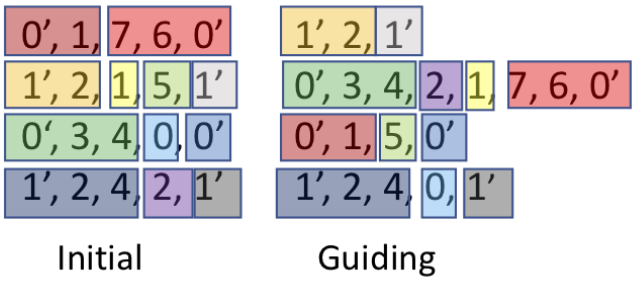

FIGURE 1: Finding segment boundaries between an initial and guiding solution

Segment difference is always greater than arc difference, as every arc difference in the two solutions must necessarily also be a segment difference. However, as shown above, segment differences can exist without a corresponding arc difference. As it is computationally more expensive to calculate segment difference and difficult to incrementally update, we choose to define solution distance using both metrics and only use segment differences when the arc sets do not allow identification of a distance improving transformation.

0.1.3 Overall Methodology

In reviewing literature, there are two different approaches to relinking solutions consisting of sets of routes. In one approach, the relinking is undertaken route-by-route with some matching process such as most similar, least similar, randomly selected, etc., defining the pairs to undergo relinking (Ho and Gendreau, 2006). The other more general approach is to relink across solutions as originally described by Glover et al. (2000).

We initially consider relinking route by route with the distance measure comprising the sum of the arc, vertex, and route differences. However, two significant issues are identified with this approach, motivating the implementation of a more sophisticated approach.

1. The neighbourhood operators and single distance measure, do not guarantee convergence to S^g
2. Relinking route-by-route limits the hybridisation potential of structurally different routes.

As a result of the shortfalls identified above, a more generalised approach of relinking whole solution sets is adopted. The neighbourhood operators transform the set of vertices visited rather than the whole of the solution (i.e., the order flow paths or the allocation of orders to stops) and retain the computationally simple arc difference distance measure supplemented with a more computationally intensive segment-based distance measure. We recognise that this approach ignores the intrinsic linking of the collection and delivery nodes, and suggest that future research on neighbourhoods operating on the order path may be fruitful.

The process of incrementally transforming the initial solution S^o through the solution space via a set of neighbourhood operators to the guiding solution S^g follows the steps described below and is presented as pseudo-code at Algorithm 2. The 'best move' is the neighbourhood operator applied to a specific vertex (or set of vertices) that provides the maximum reduction in solution distance to the guiding solution. If two options have the same resulting solution distance, the one that results in an overall lower transport cost is the best move.

- Remove any identical routes in the initial solution S^o, and guiding solution S^g. Only perform relinking across the part of the solution that has differences.
- In line 2, add degenerate routes to S^o ensuring that at least as many routes originate from each cross-dock as in S^g.
- Set δ to be $\delta^1(S^o)$, i.e., use the primary distance measurement method in the algorithms.

- Construct the sets V_i^o, A_{ij}^o and V_i^g, A_{ij}^g, $\forall i, j \in V'$, being route and stop number tuples of vertices and arcs, respectively (see Section 0.1.2).
- Set $\gamma = 1$, and $S^u = S^o$ so the algorithm starts from the initial solution.
- In lines 6–23, while $\delta(S^u) > 0$ test the operators in the order listed, selecting the first operator with a distance reducing move.
- In lines 7–16, evaluate neighbourhoods and exit the selected block when the first improving move is identified. When selecting arcs, the first element $a_{ij} \in A_{ij}^\gamma$, $i, j \in V'$, that satisfies any stated condition is chosen.
- In line 8–12, select in random sequence vertices i that have a different cardinality in the current and guiding solutions until all such vertices are tested or a distance improving move is identified,
  - Insert(i) - If more instances of i exist in the guiding solution find an insert option for i into S^u with the biggest solution distance improvement,
  - Delete(i) - otherwise, find the location to delete i from S^u that most improves the solution distance,
- Transpose(i, j) - select i, j such that $D_{ij}(S^u, S^g) > 0$ and $D_{ji}(S^g, S^u) > 0$, and find the $(r, k) \in A_{ij}^\gamma$ for which swapping the order of $\sigma_r(k)$ and $\sigma_r(k+1)$ is the best move,
- 3/4-Opt(i, j) - Select i, j such that $D_{ij}(S^u, S^g) > 0$, meaning an arc a_{ij} must be removed from the current solution to move towards the guiding solution. For the stops leading to and from each arc, a_{ij}, create segments matching sequences of stops in the guiding solution. Using the end points of these segments, locate a second arc $a_{lm} \in S^u$ such that, replacing arc a_{ij} and a_{lm} with arcs a_{im} and a_{lj} creates an arc required in the guiding solution and relocating the segments created from these two arcs results in at least one further arc required in the guiding solution. Each of the four possible recombination (one of which can be a null length segment) for each candidate i, j are tested, retaining the best move.
- 2-Opt*(a_{ij}, a_{lm}) - select elements $a_{ij} \in A_{ij}^\gamma$, $a_{lm} \in A_{lm}^\gamma$ in such a way that both $D_{ij}(S^u, S^g) > 0$ and $D_{lm}(S^u, S^g) > 0$, and either $D_{im}(S^g, S^u) > 0$ or $D_{lj}(S^g, S^u) > 0$. Splitting two routes and arcs so selected and combining the respective fronts and tails is a solution distance reducing operation.
- In line 17, apply the move identified in lines 8–16 and set $\gamma \leftarrow \gamma + 1$, reflecting the new solution.
- Lines 18–19, if no improving option is identified, switch to the secondary distance measure. This runs the segmentation process and sets $A_{ij}^\gamma = \bar{A}_{ij}^\gamma$, resulting in the algorithm only considering the arcs between segments as described in Section

0.1.2. $D_{ij}(S^u, S^g)$ now returns the difference in arcs representing segment breaks rather than the simple arc difference of distance metric δ^1.

- Lines 20–21 remove routes that are identical in the current solution and the guiding solution from both solutions.
- If no routes remain in either solution, exit from the process.
- If only degenerate routes exist in the relinked solution, remove them and exit the process.

Algorithm 2 Overall Methodology

1: Remove all routes identical in $\bar{R}(o), \bar{R}(g)$
2: Add necessary degenerate routes to $\bar{R}(o)$
3: $\delta \leftarrow \delta^1(S^u)$ ▷ Arc difference
4: Create V_i^o, A_{ij}^o and V_i^o, A_{ij}^o
5: $\gamma \leftarrow 1, S^u \leftarrow S^o$
6: **while** $\delta(S^u) \neq 0$ **do**
7: **Select** best/first distance reducing m from:
8: **for all** $i \in V'$ s.t. $D_i(S^u, S^g) \neq 0$ and m_1 not found **do**
9: **if** $D_i(S^u, S^g) < 0$ **then**
10: $m_1 \leftarrow$INSERT(i)
11: **else**
12: $m_1 \leftarrow$DELETE(i)
13: $m_2 \leftarrow$TRANSPOSE$(i, j$ s.t. $D_{ij}(S^u, S^g) > 0, D_{ji}(S^g, S^u) > 0)$
14: $m_3 \leftarrow$3/4-OPT$(i, j$ s.t. $D_{ij}(S^u, S^g) > 0)$
15: $m_4 \leftarrow$2-OPT$^*(i, j$ s.t. $D_{ij}(S^u, S^g) > 0)$
16: **EndSelect**
17: Apply m, $\gamma \leftarrow \gamma + 1$
18: **if** $\delta^1(S^u) = \delta^1(S^{\gamma-1})$ **then**
19: $\delta \leftarrow \delta^2(S^u)$ ▷ Segment difference
20: **if** $r = r'$ for $r \in \bar{R}(\gamma)$, $r' \in \bar{R}(g)$ **then**
21: remove r and r' from the solutions
22: **if** No routes or only degenerate routes in $\bar{R}(u)$ **then**
23: exit

We test performing a MILP local optimisation using the route sets from relinked solutions and find these are rarely feasible. We also test using the relinked routes and the routes from the guiding solution and rarely find the solution differs from the guiding solution. However, adding the set of all new routes created in relinking to the solution pool often results in an improving solution to the resultant MILP. As such, we choose to only evaluate the utility of routes created by path relinking once all of the route improvement processes are completed.

0.1.4 Implementation of Operators

A selection of operators are defined to incrementally transform an initial solution into the guiding solution. These operators are based on the traditional k-opt operators initially introduced by Croes (1958) and Lin and Kernighan (1973) and subsequently developed by many other authors. In our case, we are only interested in moves reducing the distance in solution space and can therefore exclude any move guaranteed to increase this distance. We establish the set of feasible moves by building a set of arcs we know to be missing in the initial solution with respect to the guiding solution, a set of arcs we know to be surplus and a set of vertices that have a different quantity in the two solutions. Because SPDVRP-CD admits solutions that revisit a vertex an arbitrary number of times, it is common for a vertex to appear a different number of times in the two solutions.

The first two operators, vertex insert and delete, act to bring the total number of vertices in the two solutions into alignment. These are normally distance reducing moves, as a vertex difference causes at least 3 arcs to be different. Making a vertex change fixes at least one of the arc differences, although the delete operation can in some cases also create a new arc difference. It is clear reversing any sequence existing in both solutions must be distance increasing. Therefore, we implement non-inverting variants of k-opt that swap segments within a route, across two routes, or split two routes and recombine heads and tails. Finally, we include a paired vertex transpose, that swaps position of two adjacent vertices out of order with respect to each other as the only operator that inverts order. With these operators, the degenerate routes, and the degenerate route clean-up process, we are able to transform any initial solution to a guiding solution.

Insert Vertex

The insert vertex neighbourhood identifies the route and stop number in S^u where inserting a vertex $i \in V'$ provides the largest reduction in δ at the lowest incremental cost. If a vertex i is missing, its connecting arc $a = (i, v^+)$, v^+ being the next stop, must also be missing. Similarly arc (v^-, i), v^- being the prior stop, must also be missing from S^u. Below the process of identifying where to insert the vertex i is described and pseudo-code is presented in Algorithm 3,

1. Bestmove is a tuple containing ((r,k), distance, incremental cost), representing a specific candidate arc into which i could be inserted, the resulting solution distance and incremental transport cost of making the insertion. In line 2 bestmove is initialised as a null move.

2. For each j such that $D_{ij}(S^g, S^u) > 0$, $j \in V'$ (i.e., the guiding solution has more arc from i to j than the current solution),

3. For each arc $(r,k) \in A^{\gamma}_{pj}$ s.t. $D_{pj}(S^u, S^g) > 0$, $p \in V'$ (i.e., the current solution has more arcs from p to j than the guiding solution)

4. Lines 5–10 test the solution distance improvement and incremental cost change of inserting vertex i between $\sigma_r(k) = p$ and $\sigma_r(k+1) = j$.

5. Lines 7–9 test if the solution distance improvement is greater than any previously found insert location, or the solution distance improvement is the same and the incremental cost is lower. If this is the case, then bestmove is updated with the current (r,k), solution distance improvement, and incremental cost.

6. Lines 9–10 identify if the specific insertion of i at (r,k) is required (that is the sequence $(\sigma_r(k), i, \sigma_r(k+1))$ in the guiding solution is missing from the current solution) and, if so, exits the process returning this move.

7. The whole process is then run similarly for arcs ending at i.

8. Finally, if no bestmove is identified, this indicates three or more vertices need to be added as a segment, and i is an internal vertex in the segment. Once part of the segment leading to i is added by another operator, an insertion of i with improving solution distance will be possible.

Algorithm 3 Insert Neighbourhood

```
1:  function INSERT(i)
2:      bestmove ← (none, 0, ∞)
3:      for all j s.t. D_ij(S^g, S^u) > 0, j ∈ V' do
4:          for all (r,k) ∈ A^γ_pj s.t. D_ij(S^u, S^g) > 0, p ∈ V' do
5:              δ ← distance after replacing (σ_r(k), σ_r(k+1)) with (σ_r(k), i, σ_r(k+1))
6:              cost ← I(σ_r(k), i, σ_r(k+1)) − I(σ_r(k), σ_r(k+1)))
7:              if δ <bestmove(2) or (δ = bestmove(2) and cost < bestmove(3)) then
8:                  bestmove ← ((r.k), δ,cost)
9:                  if δ = 3 then
10:                     return bestmove
11:     Execute the above for arcs leading to i
12:     return bestmove
```

Delete Vertex

The delete vertex neighbourhood removes a surplus vertex from the current solution using the process shown in Algorithm 4 and described below.

1. Line 2 initialises bestmove as a tuple representing a null move, (None, 0, ∞).

2. For each each (r, k) in the vertex set V_i^γ.
3. In Line 4 if (r, k) is either the start or end of an arc that exists in the current solution and does not exist in the guiding solution, then evaluate the benefit of removing (r, k) from the current solution.
4. If $\sigma_r(k) \in V'_x$ and the stop at $k + 1$ is also in the origin/terminus set, then this is a vertex in a degenerate route (having only the origin and terminus stops) and is preferentially selected for removal resulting in the deletion of both vertices and the excess degenerate route.
5. Lines 7–8 calculate the distance saving and incremental cost saving of removing (r, k) from the current solution.
6. Lines 9–10 test if the distance improvement is greater than any previously found delete location, or the distance improvement is the same and the incremental cost is lower, and if so updates the bestmove with the current (r, k), distance improvement, and incremental cost.

Algorithm 4 Delete Neighbourhood

```
1:  function DELETE(i)
2:      bestmove = (none, 0, inf)
3:      for all (r, k) ∈ V_i^γ do
4:          if D_{σ_r(k),σ_r(k+1)}(S^u, S^g) > 0 or D_{σ_r(k−1),σ_r(k)}(S^u, S^g) > 0 then
5:              if σ_r(k) ∈ V'_x and σ_r(k + 1) ∈ V'_x then
6:                  return ((r, k), 1, 0)                    ▷ Degenerate route,
7:              δ ← reduction for removing σ_r(k) from (σ_r(k − 1), σ_r(k), σ_r(k + 1))
8:              cost ← I(σ_r(k − 1), σ_r(k + 1)) − I(σ_r(k − 1), σ_r(k), σ_r(k + 1))
9:              if δ <bestmove(2) or (δ = bestmove(2) and cost < bestmove(3)) then
10:                 bestmove ← ((r, k,δ,cost)
11:     return bestmove
```

It is possible for there to be no distance reducing result from the Delete operator. Consider $S^u = (1, 3, 2, 5, 4, 2, 1)$ and $S^g = (1, 5, 4, 3, 2, 1)$. $v = 2$ is surplus and the extra arcs in the current solution compared to the guiding solution using $v = 2$ are $(2, 5), (4, 2)$. No missing arcs will be added by deleting either instance of $v = 2$. Deleting $\sigma_r(6)$ increases distance as the operation removes the extra arc $(4, 2)$, but adds a new extra arc $(4, 1)$ and removes the required arc $(2, 1)$, deleting $\sigma_r(3)$ removes the extra arc, $(2, 5)$, but adds a new extra arc, $(3, 5)$, with no net change in $\delta(S^u)$. In this case, Delete(2) would not be on the relink path, until after 3 was relocated using a k-Opt move.

Another possible case is $S^u = (1, 3, 2, 1), (1, 2, 4, 1)$ and $S^g = (1, 3, 2, 4, 1)$. The combination of removing a 2 and a 2-opt exchange creates the S^g route and a degenerate $(1, 1)$

route. As a design decision, this is handled by first applying a 2-opt moving vertex 3 into sequence and then deleting 2 from the resulting short route, creating a degenerate route, which is then removed.

Transpose Arcs

Vertices i, j are passed and all arcs connecting i, j are tested to identify the best arc to transpose. The process is described below and shown in Algorithm 5.

1. Line 2 initialises bestmove as a tuple of ((r, k), distance, incremental cost) to a null move.
2. Select each $(r, k) \in A_{ij}^{\gamma}$ that is the first stop of arcs in the form of a route stopping at i followed by j.
3. Lines 4–5 calculate the distance saving and incremental cost saving of swapping the stops at k and $k + 1$ in r from the current solution.
4. Lines 6–8 test if the distance improvement is greater than any previously found transpose option, or the distance improvement is the same and the incremental cost is lower, and updates the bestmove with the current (r, k), distance improvement, and incremental cost.

Algorithm 5 Transpose Neighbourhood

```
1: function TRANSPOSE(i, j)
2:     bestmove = (none, 0, inf)                    ▷ Initialise best move found to None
3:     for all (r, k) ∈ A_ij^γ do
4:         δ = distance reduction replacing σ_r(k), σ_r(k+1) with σ_r(k+1), σ_r(k)
5:         cost ← I((..., σ_r(k−1), σ_r(k+1), σ_r(k), σ_r(k+2), ...)) − I(r)
6:         if δ <bestmove(2) or (δ = bestmove(2) and cost < bestmove(3)) then
7:                                  ▷ Improved distance, or same distance for lower cost insert
8:             bestmove ← ((r, k), δ,cost)
9:     return bestmove
```

3/4-Opt

This neighbourhood operation interchanges two segments from the same or different routes, at most one segment being a null segment. The segment construction definition excludes the origin and terminus stops from segments. Swapping segments including an origin or terminus requires additional constraints to avoid creating illegal routes and is documented as 2-Opt*.

Two support functions are defined. The first creates maximal sequences of stops existing in both the current and guiding solutions, called segments, and the second identifies arcs missing in the current solution, that could be created by swapping segments in the current solution.

CreateSegment is provided and builds two segments by adding adjacent vertices going forward and backward in the route, until the next addition would create a segment that does not exist in S^g. The segment construction is ended prior to the terminus or origin vertex, resulting in segments defined by two arc bounding arcs, or one bounding arc and a zero length segment for a segment starting or ending adjacent to a terminus/origin. This process is similar to that described in Section 0.1.2 and the implementation is shown in Algorithm 6. Starting with the first stop of an arc a_{ij} in a route, build a segment stepping towards, but not including, the origin until a matching segment doesn't exist in the guiding solution. Similarly, build a sequence from the stop at j towards, but not including, the terminus until a matching segment doesn't exist in the guiding solution. The function returns a pair of tuples with each tuple containing the route and stop number after which the segment break forming each end of the segment is placed.

MatchingArcs using a template arc, a_{ij}, locates arcs in S^u, that when removed, would allow a recombination that adds an arc in S^g currently missing from S^u.

- Line 11 initialises result, which will be a set of all candidate arcs that are candidates for a 3/4 opt operation including a_{ij}.
- All of the arc sets with a greater cardinality in the guiding solution and which share their starting location l with the ending location j or share an ending location m with the starting location i are found.
- Lines 14 and 15 test each of these candidate arcs against another set of arcs $a_{\alpha\beta}$ for arcs that are surplus in the existing solution and start at location l or finish at location m. Arcs satisfying these conditions are candidates for a successful 3/4 opt operation and are added to the result set.
- In line 16 a random element from the result set is returned.

Algorithm 6 3/4 Opt Support Functions

function CREATESEGMENT(a_{ij}) ▷ a_{ij} refers to an (r, k)
  for $k' = k$ downto 2 **do**
    if $(\sigma_r(k'-1), \ldots, \sigma_r(k)$ doesn't exist in S^g **then** Exit Loop
  seg1 $\leftarrow ((r, k'), (r, k))$
  for $k' = k+1$ to $n_r - 1$ **do**
    if $\sigma_r(k+1), \ldots, \sigma_r(k'+1))$ doesn't exist in S^g **then** Exit Loop
  seg2 $\leftarrow ((r, k+1), (r, k'))$
  return (seg1, seg2)

function MATCHINGARC(a_{ij})
  result $\leftarrow \emptyset$
  for all a_{lm} s.t. $D_{lm}(S^g,S^u) > 0$ and $(l = j$ or $m = i)$ **do**
    for all $a_{\alpha\beta}$ s.t. $D_{\alpha\beta}(S^u, S^g) > 0$ **do**
      if $D_{\alpha\beta}(S^u, S^g)$ s.t. $\alpha = l$ or $\beta = m$ **then**
        result $\leftarrow a_{\alpha\beta} \cup$ results
  return random element from result

2 shows the construction of the candidate swap segments and then the application of the 3/4-Opt operation.

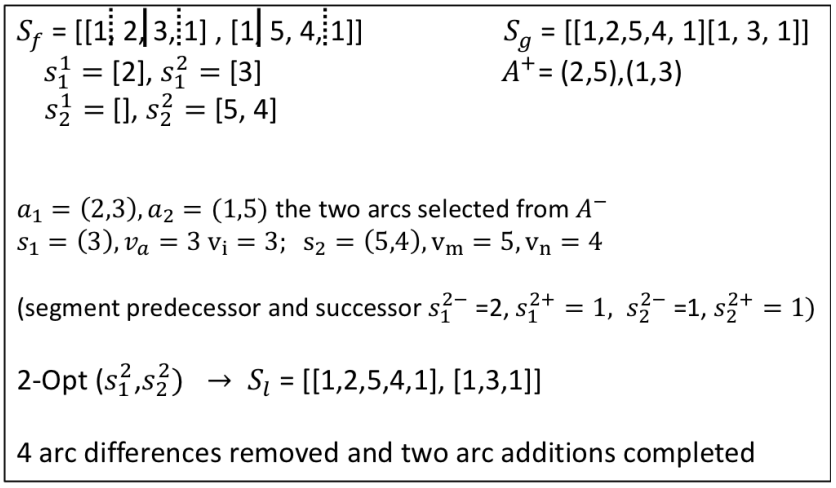

FIGURE 2: 3/4-Opt operation

The process for undertaking a 3/4-Opt is described below with pseudo-code in Algorithm 7.

1. Vertices i and j are provided to the function, selected such that the cardinality of the arc set for A_{ij}^{γ} exceeds that of the arc set for A_{ij}^{g}.

2. Line 2 initialises bestmove as a tuple of (segments to swap, distance, incremental cost).

3. For each $a_{ij} \in A_{ij}^{\gamma}$ in lines 4-16 four 3/4 Opt strategies are tested based on the a_{ij} arc.

4. In lines 4–7 an arc b_{lm} is selected using the MatchingArc function exposing a vertex pair l, m that could be distance improving if connected to the locations i and j. The segments bounding a_{ij} and b_{lm} are identified using the CreateSegment function. Seg_1^a is the stop defining the start of the segment leading up to and including a_{ij} (i.e. including vertex i) and seg_2^a is the stop defining the end of the segment starting at $\sigma_r(k+1) = j$. Seg_1^b and seg_2^b are similarly defined for b_{lm}.

5. Line 7 assigns the route element of the tuples a_{ij} and b_{lm} to r_a and r_b respectively.

6. Line 8 assigns in turn to α and β the segment exchange patterns to be used in testing the 3/4 opt exchanges.

7. In lines 9–17, each of the four possible exchanges of the 4 identified segments is evaluated. Note, it is possible a segment is a null segment and hence the exchange is a null operation (causing a 3 opt instead of a 4 opt exchange).

8. Lines 9 and 10 build temporary routes using the current exchange pattern defined in line 8 to exchange a segment from r_a with a segment from r_b.

9. Lines 11 and 12 determine the solution distance reduction and the incremental cost change of implementing the exchange.

10. Lines 13–15 test if the distance improvement is greater than any previously found transpose option, or the distance improvement is the same and the incremental cost is lower, and updates the bestmove with the current segment swap, distance improvement, and incremental cost.

11. Execute lines 3–15 for arcs leading to i (A_{ji}^{γ}).

Algorithm 7 3/4 Opt Neighbourhood

function 3/4Opt(i, j)
  bestmove = (none, 0, inf) ▷ Initialise best move found to None
  for all $a_{ij} \in A_{ij}^{\gamma}$ **do**
    $b_{lm} \leftarrow$ MatchingArc(a_{ij})
    seg_1^a, seg_2^a = CreateSegment(a_{ij})
    seg_1^b, seg_2^b = CreateSegment(b_{lm})
    $r_a \leftarrow$ from a_{ij}, $r_b \leftarrow$ from b_{ij},
    for $(\alpha, \beta \in \{(1,1),(1,2),(2,1),(2,2)\}$ **do**
      $t_a \leftarrow r_a$ with seg_α^a replaced with seg_β^b
      $t_b \leftarrow r_b$ with seg_β^b replaced with seg_α^a
      δ = distance reduction from replacing r_a, r_b with t_a, t_b
      cost $\leftarrow I(t_a) + I(t_b) - I(r_a) - I(r_b)$
      if $\delta <$bestmove(2) or ($\delta =$ bestmove(2) and cost $<$ bestmove(3)) **then**
        ▷ Improved distance, or same distance for lower cost insert
        bestmove $\leftarrow ((t_a, t_b), \delta$,cost)
    Execute the above for arcs leading to i
  return bestmove

2-Opt*

A special case exists where the segments being swapped are bounded by the start or terminus of the route. To avoid creating illegal routes, 2-Opt* only is considered when both candidate routes are based at the same cross-dock.

2-Opt* is considered when two arcs a_{ij}, a_{lm} s.t. $D_{ij}(S^g, S^u) > 0$ and $D_{lm}(S^g, S^u) > 0$ and at least one of $D_{jl}(S^g, S^u) > 0$ or $D_{mi}(S^u, S^g) > 0$ are true. This set of conditions implies there are two arcs in the current solution that must be removed, and joining at least one of the vertices defining arc a_{ij} to one of the vertices of arc a_{lm} creates an arc that is in the guiding solution and not in the current solution. If a_{ij}, a_{lm} are in different routes both originating from the same cross-dock, then the 3/4-Opt process is executed except the segments created are from the origin to the arc and the arc to the terminus so the swap is executed for the full head and tail segments. This will reduce distance by 2 if only one of $D_{lj}(S^g, S^u) > 0$ or $D_{mi}(S^u, S^g) > 0$ is true and four if both are true. 3 shows an example 2-Opt* move.

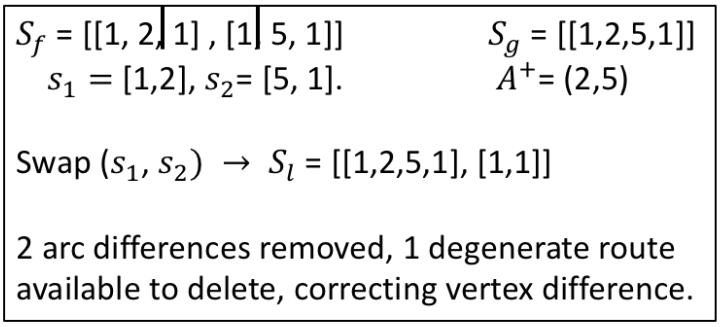

FIGURE 3: 2-Opt* operation

0.2 Definition of SPDVRP-CD

We introduce a variation on the vehicle routing problem to address the business challenges of Less than Truck Load (LTL) networks with transfer points, an important supply chain structure for industry. This problem has a set of *order requests*, to be collected from locations designated as *suppliers* and delivered to *delivery* locations. Additionally, there is a set of locations at which vehicles are based and where orders can be transferred, split or consolidated. The transfer locations are called cross-docks and it is common for some delivery locations to be used as cross-docks.

This physical structure can be represented as a set of vertices in a graph, where the arcs represent the connections available between locations. Each arc is labelled with a cost and elapsed time. The elapsed time includes the travel time and the service time at the departure location. The arcs are consistent with the triangle inequality rule, which requires it always be at least as quick and no more expensive to go directly between two vertices rather than via a third vertex.

We formally define the locations as follows:

V_s is the set of all supplier locations,

V_d is the set of all delivery locations (customers, potentially including cross-docks),

V_x is the set of all cross-dock locations, which may include some locations $v \in V_\mathrm{d}$ requiring a delivery,

$V = V_\mathrm{s} \cup V_\mathrm{d} \cup V_\mathrm{x}$ is the set of distinct locations within the network.

We define a graph composed of the distinct vertices V connected by all the arcs available for use, as well as two supporting matrices describing the cost and time to use each arc:

$G = (V, A)$ is a directed graph of vertices V and arcs A,

$C = (c_{ij})$ is a matrix defining the travel costs for each arc (i, j),

$T = (t_{ij})$ is a matrix defining the travel time for each arc(i, j).

Each supplier can have an order request for multiple delivery location, and similarly each delivery location may have multiple suppliers. An order is a requirement to move a number of pallets from a supplier location $s \in V_\mathrm{s}$ to a delivery location $d \in V_\mathrm{d}$, represented as (s, d). Each order request has an associated volume q_{sd}, measured in pallets, an earliest collection time e_{sd}, and a latest delivery time l_{sd}. The supply chain normally has a short time between earliest collection and the latest delivery of an order request and does not require an earliest time for the delivery. However, the conversion to

delivery windows is a trivial extension, implemented by defining a latest collection time and earliest delivery time. These variables are then incorporated into the formulation analogously to e_{sd} and l_{sd}.

O= $\{(s,d)|q_{sd} > 0\}$ is the set of order requests to satisfy,

q_{sd} is the number of pallets to be moved between s and d,

e_{sd} is the earliest time request (s,d) can be collected from s,

l_{sd} is the latest time request (s,d) can be delivered to d.

The order requests are transported on vehicle routes reflecting the path the driver takes, rather than the paths of the trailer or tractor. The operational detail of transferring a load from one vehicle to another via a transshipment, swapping trailers, or drivers swapping vehicles is generalised as product transferring from one route to another at a cross-dock. All vehicles are standard size articulated vehicles with capacity Q.

A route is defined as a closed walk in G, originating and terminating at a vertex of V_{x}. Specifically, it must start and terminate at the same cross-dock, can visit suppliers, delivery points and other cross-docks any number of times, and must have an overall travel time that does not exceed the maximum allowed driver shift time. If a route visits a cross-dock $v \in V_{\mathrm{x}}$ other than the origin/departure, v is duplicated in the route representing the arrival/unloading and the loading/departure activities separately and allowing a delay between arrival and departure to synchronise with product arriving on other routes.

A vehicle route r is represented as a sequence of n_r stops, as defined by

$$r = (\sigma_r(1), \ldots, \sigma_r(n_r)),$$

where $\sigma_r(1) = \sigma_r(n_r) \in V_{\mathrm{x}}$ and $\sigma_r(k) \in V$, $\forall k \in \{2, \ldots, n_r - 1, \}$. If the kth stop is a vertex (i.e. $\sigma_r(k) \in V_{\mathrm{x}}$) then, for computational convenience, that vertex is replicated in the route (i.e. $\sigma_r(k) = \sigma_r(k+1)$). In this case, we interpret $\sigma_r(k)$ to be the arrival and unloading at the cross-dock and $\sigma_r(k+1)$ to be the loading of pallets for subsequent delivery. Separating the arrival and departure into two separate stops at a cross-dock facilitates the introduction of a delay between the arriving and unloading of pallets and the subsequent loading and departure of pallets for delivery to later stops. This allows vehicles to wait in the cross-dock to load pallets that arrive after the time of arrival at $\sigma_r(k)$ and before the departure time from $\sigma_r(k+1)$. Each route has a maximum duration, $t_{\max}$ and a cost, $\bar{c}_r$, being the sum of the arc costs,

$$\bar{c}_r = \sum_{k=1}^{n_r-1} c_{\sigma_r(k),\sigma_r(k+1)}, \qquad \forall r \in R. \tag{1}$$

A vehicle may wait for a collection window, or wait at a cross-dock for an order to be delivered to the cross-dock on some other route. w_{rk} $\forall r \in R$, $k \in \{2, \ldots, n_r - 1\}$ is an integer variable reflecting the waiting time, c^{w} is a constant representing the cost per minute for a vehicle waiting. The total cost of a solution is therefore the sum of the travel cost for all of the routes used (Equation 1), plus the total of waiting costs defined by,

$$\text{Total waiting cost} = \sum_{k=2}^{n_r-1} c^{\mathrm{w}} w_{rk}, \qquad \forall r \in R. \tag{2}$$

A solution of the SPDVRP-CD is a set of routes and an allocation of requests or fractions of requests to these routes that minimise the total cost, subject to:

- each request being completed within the time constraints,
- the vehicle's load never exceeds the capacity of the vehicle,
- each request can be loaded or unloaded on a route only at its supply, destination, or cross-dock,
- a request can only be loaded at a cross-dock if, prior to the time of loading, the quantity to be loaded has been delivered and not yet loaded at a time earlier than the departure time of the route from the cross-dock,
- order requests follow a path through G which is time feasible, i.e. for each route r, the arrival time at $\sigma_r(k)$ is no earlier than the time at $\sigma_r(k-1)$ plus the travel time $t_{\sigma_r(k-1),\sigma_r(k)}$, for $k \in \{2, \ldots, n_r\}$. For simplicity in the MILP formulation, service time is assumed to be fixed and included in the travel time and driver break requirements are ignored.

The cost function includes the transit cost c_{ij} for each arc (i, j) used in the routes within the solution, the cost of any waiting at a vertex for a collection time or for product to arrive for cross-docking, and optionally a fixed cost for each route and a variable transshipment cost. The variable transshipment cost is a cost per pallet unloaded from one vehicle and loaded unto another vehicle. It is assumed pallets that arrive and depart on the same vehicle are loaded in such a way they remain on the vehicle and do not incur a transshipment cost.

The model parameter notation and brief definitions are summarised in Table 1 .

Symbol	Description
V_{s}	Set of supplier locations
V_{d}	Set of delivery locations
V_{x}	Set of locations at which vehicles are based and cross-docking can occur.
V	Set of distinct locations.
G	The directed graph of vertices V with connecting arcs A
c_{ij}	The transit cost for the arc between $i \in V$ and $j \in V$
t_{ij}	The transit time for the arc between $i \in V$ and $j \in V$
q_{sd}	Number of pallets to be moved between s and d
e_{sd}	Earliest time request (s, d) can be collected from s
l_{sd}	Latest delivery time for (s, d)
O	Set of orders

TABLE 1: Model parameters

Bibliography

Croes, G.A., 1958. A method for solving traveling-salesman problems. Operations Research 6, 791–812.

Glover, F., Lagunai, M., Marti, R., 2000. Fundamentals of scatter search and path relinking. Control and Cybernetics 29, 653–684.

Ho, S.C., Gendreau, M., 2006. Path relinking for the vehicle routing problem. Journal of Heuristics 12, 55–72.

Huang, Y.H., Ting, C.K., 2011. Genetic algorithm with path relinking for the multi-vehicle selective pickup and delivery problem. 2011 Ieee Congress on Evolutionary Computation (Cec) , 1818–1825.

Lin, S., Kernighan, B.W., 1973. Effective heuristic algorithm for traveling-salesman problem. Operations Research 21, 498–516.

Prins, C., 2004. A simple and effective evolutionary algorithm for the vehicle routing problem. Computers & Operations Research 31, 1985–2002. URL: <GotoISI>://WOS:000222247600003, doi:10.1016/S0305-0548(03)00158-8. 832do.