UCLouvain

École polytechnique de Louvain

Insertion Sequence Variables in MiniCP

Author: **Quentin DELMELLE**
Supervisor: **Pierre SCHAUS**
Readers: **Hélène VERHAEGHE , Charles THOMAS**
Academic year 2021–2022
Master [120] in Computer Science and Engineering

Abstract

The Insertion Sequence Variable or ISV is a recently developed high-level dynamic data structure used for modeling sequences of events in constraint-programming environments. It is designed to be used as a tool for solving CP-based scheduling and routing problems. LNS-FFPA is a state of the art search algorithm for the Dial-A-Ride-Problem or DARP, a well known routing problem with many real life applications. The goal of this paper is to study the impact of the introduction of ISVs to replace regular sequence models in a LNS-FFPA solver for the DARP. To do this, 2 implementations of LNS-FFPA have been developed in MiniCP, a lightweight open-source CP environment. One uses ISVs while the other uses a regular successor array. Experimental results have shown that ISVs do not improve performance when used in a strictly implemented LNS-FFPA solver, but some modified versions of LNS-FFPA using ISVs might be promising.

Contents

Chapter 1

Introduction

The aim of this paper is to implement a LNS-FFPA solver for the DARP in MiniCP using insertion sequence variables (ISVs) and compare it with a more standard LNS-FFPA solver, with the intent of producing experimental results that can indicate if the use of ISVs has an overall impact on the performances of the solver.

The DARP (Dial A Ride Problem) is a well known constrained optimization problem that consists in finding an optimal route for a fleet of vehicles with limited capacity, which have to pick up and drop customers at several points while minimizing the total distance traveled.

LNS-FFPA is a state-of-the-art search algorithm for DARP solvers. It was introduced by Siddhartha Jain and Pascal Van Hentenryck in 2011 [1].

MiniCP [2] is constraint programming environment in java designed by Laurent Michel, Pierre Schaus and Pascal Van Hentenryck in 2021 for teaching purposes.

The Insertion Sequence Variable [3] is a data structure designed by C. Thomas, R. Kameugne and P. Schaus in 2020. Its purpose is to model sequences of events in CP environments.

Chapter 2

Context

2.1 The Dial a Ride Problem

The Dial-a-ride problem or DARP is an iconic problem in the field of constrained optimization. It consists of finding an optimal route for multiple vehicles with limited capacity, which have to pick up and drop customers at several points scattered over a 2D space. It is a sequence-based problem, which means we have to schedule a series of tasks (or requests) in an optimal order, according to limited resources and time windows. This problem is known to be NP-Complete. There are many different approaches and formulations for this problem. The following formulation is based on the DARP as described in [1]. It is a static DARP, where all requests are known in advance and all vehicles start and end their routes at a central depot. The goal is to predict the optimal itinerary of each vehicle.

2.1.1 Input

The input of a DARP instance consists of m vehicles, n requests, the maximum ride time L for customers, the maximum route duration D for any vehicle and the time horizon T, assuming that the starting time is 0. Each vehicle k has a maximum load Q_k. The DARP is defined on a complete graph $G := (V, E)$ where $V = \{v_0, v_1, ...\}$ is the set of vertices and $E\{(v_i, v_j) : v_i, v_j \in V, i \neq j\}$ the set of edges. The distance between any pair of vertices v_a and v_b, defined as $dist_{a,b}$ is known in advance.

A request $i \in \{0..n-1\}$ is defined as a pair of vertices, each vertex v_j being associated with a serving duration d_j, a load q_j, and a time window $[e_j, l_j]$. We define the pickup vertex as v_i, and the delivery vertex as v_{i+n}. This means that there are $2n$ sites that must be visited. In addition to these sites, each vehicle $k \in \{0..m-1\}$ has a start depot v_{2n+k} and an end depot v_{2n+m+k}. We also have

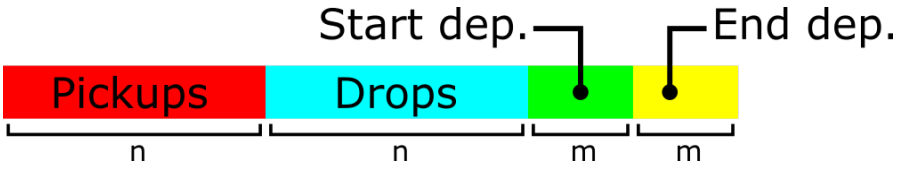

Figure 2.1: vertex numbering system

$d_x = 0$, $q_x = 0$, and $[e_x, l_x] = [0, T]$ for all $x \geq 2n$. Note that for each request i, $q_i > 0$ and $q_{i+n} = -q_i$ (embarking at v_i, disembarking at v_{i+n}).

Each request can be either *inbound* or *outbound*. If a request j is inbound, the delivery vertex has window $[e_{j+n}, l_{j+n}] = [0, T]$ and the pickup vertex is *critical*, meaning it has a smaller time window $e_j > 0, l_j < T$. If a request is outbound, then the pickup vertex has window $[e_j, l_j] = [0, T]$ and the delivery vertex is *critical*, i.e. $e_{j+n} > 0, l_{j+n} < T$.

2.1.2 Output

A solution to the DARP is defined by a set of sequences, one for each vehicle, representing the vertices it will visit in the right order.

$$\forall k \in \{0..m-1\}, \vec{R_k} := \{v_{k1}, v_{k2}, ..., v_{kx}\} \tag{2.1}$$

Each vertex must be visited exactly once by one of the vehicles, which implies:

$$\vec{R_0} \cup \vec{R_1} \cup ... \cup \vec{R_{m-1}} = V \tag{2.2}$$

$$\forall a, b \in \{0..m-1\} | a \neq b, \vec{R_a} \cap \vec{R_b} = \emptyset \tag{2.3}$$

Figure 2.2 is an example of a DARP instance with $n = 3$ and $m = 2$. All the depots $v_6 \rightarrow v_9$ are at the same location $(0, 0)$. The represented solution is $\vec{R_0} = \{v_6, v_2, v_1, v_4, v_5, v_8\}, \vec{R_1} = \{v_7, v_0, v_3, v_9\}$.

Vertex	x	y	e	l	d	q
0	3	2	0	60	5	2
1	5	0	10	20	5	1
2	2	-1	0	60	5	1
3	-2	4	10	20	5	-2
4	2	-4	0	60	5	-1
5	-4	-2	40	50	5	-1

Vehicle	Capacity
0	2
1	2

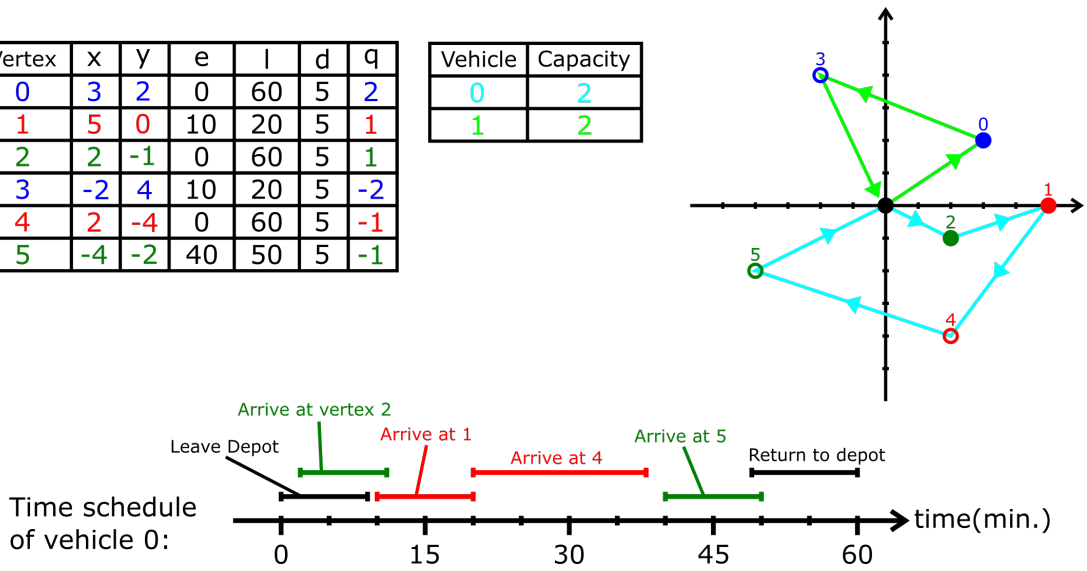

Figure 2.2: Example DARP with solution

2.1.3 Constraints

Let's define b_j as a variable representing the serving time of vertex v_j (i.e. the instant a vehicle arrives at v_j). It is important to understand that this is not the same as the actual serving time, which would be the time the client is picked up or dropped. Vehicles do not necessarily serve a request as soon as they arrive at a vertex, there can be some waiting time.
Knowing that, the objective is to find a route $\vec{R_k}$ for each vehicle k satisfying all the following constraints, and ensuring that the total distance traveled by all vehicles is minimized.

For every vertex $j \in 0...2n-1$:

- $e_j \leq b_j \leq l_j$ **(Time Window)**

For every request $i \in 0...n-1$:

- v_i and v_{i+n} both appear in the same route **(Dependency)**
- v_i is visited before v_{i+n} **(Precedence)**
- $b_{i+n} - b_i - d_i \leq L$ **(Maximum Ride Time)**

For every vehicle k:

- $\vec{R_k}$ starts with $v_{k1} = v_{2n+k}$ and ends with $v_{kx} = v_{2n+m+k}$ **(First & Last)**

- $b_{2n+m+k} - b_{2n+k} \leq D$ **(Maximum Route Duration)**
- The total load of the vehicle never exceeds Q_k **(Cumulative)**

Also:

- If v_j directly follows v_i in a route, then $b_j \geq b_i + d_i + dist_{i,j}$ **(Transition Times)**

2.2 The LNS-FFPA Algorithm

In [1], Siddhartha Jain and Pascal Van Hentenryck explored and compared several search algorithms for the DARP such as tabu search or Variable Neighbourhood Search, and they showed that their Large Neighbourhood Search variant called LNS-FFPA was the most successful.
LNS-FFPA stands for Large Neighbourhood Search with First Feasible Probabilistic Acceptance. This means that unlike the standard LNS which searches for an improving solution in the neighbourhood of the current solution at every iteration, LNS-FFPA accepts the first feasible solution it finds when solving the sub-problem, and lets the heuristic guide the search towards better solutions.
LNS-FFPA consists of 3 main algorithms :

2.2.1 Algorithm 1: TreeSearch

This is the algorithm used for solving the sub-problem, which consists in building a complete feasible solution given a partial solution. It is a classic depth-first search that recursively chooses new unassigned requests according to a selection heuristic and tries to insert them into the routes at the optimal insertion points.

An insertion point pt for a request r is defined as a tuple $< v, p, d, e >$ where v is the vehicle/route it will be inserted in, v_p is the stop preceding the pickup vertex v_r in the partial solution, v_d is the stop preceding the delivery vertex v_{r+n} and e is the insertion cost metric, defined as

$$e(r, pt) = \alpha * costIncrease - \beta * slack. \tag{2.4}$$

$costIncrease$ is the distance added to the route due to the insertion of r and $slack$ is the slack after insertion, which is defined as the sum of the gaps of the pickup and delivery vertices. The gap gap_i of a vertex v_i inserted between v_p and v_s is defined as:

$$gap_i = max(b_s) - min(b_p) - dist_{p,i} - d_p - dist_{i,s} - d_i \tag{2.5}$$

In other words, gap_i is the difference between the maximum amount of time the vehicle has to go from v_p to v_s through v_i and the time the vehicle actually spends working during this part of its route. Remember that the serving times b_x are variables, they do not have a single value and should be thought of as intervals in time.
α and β are meta-parameters, their optimal values will be determined experimentally in chapter 4.

Algorithm 1: Algorithm 1: TreeSearch(*PartialSolution*)

```
1  if no unassigned requests left then
2  |  return PartialSolution
3  end
4  r ← GetUnassignedRequest()
5  for all feasible insertion points p for r in increasing order of e(r, p) do
6  |  Insert r at point p in PartialSolution
7  |  ret = TreeSearch(PartialSolution)
8  |  if ret is a complete solution then
9  |  |  return ret /* Feasible solution found in sub-branch     */
10 |  end
11 |  Remove r from PartialSolution
12 end
13 return False /* No feasible solution found for this sub-branch
       */
```

TreeSearch (alg.1) starts from an partial solution and works in 2 steps: first, find an unassigned request (line 4). Then, try all the feasible insertion points for that request in increasing order of the cost function e and recursively call *TreeSearch* on the modified solution (lines 5-7).

If an insertion leads to a new complete feasible solution, this solution is returned (lines 8-9). The key here is that *TreeSearch* will always return the first complete feasible solution it finds.

2.2.2 Algorithm 2: GetUnassignedRequest

This is the selection heuristic mentioned above (alg.2). Its role is to tell *TreeSearch* which request to insert and where. In short, we find among all unassigned requests the one that can be assigned to the smallest number of routes/vehicles, has the fewest insertion points and whose best insertion point has the lowest insertion cost.

Algorithm 2: Algorithm 2: GetUnassignedRequest()

1 $S_1 \leftarrow \{r : r$ is an unassigned request and the number of routes in which r can be inserted is minimized$\}$
2 $S_2 \leftarrow \{r : r \in S_1$ and the number of insertion points for r is minimized$\}$
3 $S_3 \leftarrow \{r : r \in S_2$ and the best insertion point for r increases $e(r, p)$ by the least amount$\}$
4 **return** a randomly chosen element from S_3

One important thing to note here is that all the insertion points of all the unassigned requests must be computed in advance for *TreeSearch* and *GetUnassignedRequest* to work. the details of this will be explained in chapter 3.

2.2.3 Algorithm 3: MinimizeRoutingCost

This is the main search algorithm to minimize the routing cost. It takes 6 arguments:

- s is an initial feasible solution to the DARP.
- $maxSize$ is an upper bound on the number of requests that can be relaxed.
- $range$ is used to gradually increase the neighbourhood size, i.e. the number of requests relaxed.
- $numIter$ is the number of iterations for each neighbourhood size.
- $timeLimit$ is used to stop the LNS after a given amount of time and return the best solution found.
- $0 \leq d \leq 1$ is the probability of accepting a solution with higher cost.

MinimizeRoutingCost (alg.3) repeatedly relaxes the current solution by removing some randomly selected requests from the routes (line 9) and re-builds it using *TreeSearch* over increasingly large neighborhoods (line 10) to find better solutions. The initial solution s is also found using *TreeSearch*. Every new solution found is accepted as the new current solution either if it is better than the current one, or with some probability d (lines 11-13). In this context, $f(s)$ is the cost of the solution s, i.e. the accumulated distance traveled by all vehicles in s.

Note the loop reset at lines 4-6. This is done so that the search only stops when the $timeLimit$ is reached.

Algorithm 3: Algorithm 3: MinimizeRoutingCost($s, maxSize, range, numIter, timeLimit, d$)

```
1  best ← s
2  current ← s
3  for i ← 2; i ≤ maxSize − range; i ← i + 1 do
4  |  if i = maxSize − range then
5  |  |  i ← 2
6  |  end
7  |  for j ← 0; i ≤ range; j ← j + 1 do
8  |  |  for k ← 0; k ≤ numIter; k ← k + 1 do
9  |  |  |  RelaxedSolution ← Randomly select i + j requests and remove
   |  |  |   them from current
10 |  |  |  new ← TreeSearch(RelaxedSolution)
11 |  |  |  pr ← random number between 0 and 1
12 |  |  |  if f(new) < f(current) OR pr < d then
13 |  |  |  |  current ← new
14 |  |  |  |  if f(current) < f(best) then
15 |  |  |  |  |  best ← current
16 |  |  |  |  end
17 |  |  |  end
18 |  |  |  if timeLimit reached then
19 |  |  |  |  return best
20 |  |  |  end
21 |  |  end
22 |  end
23 end
```

2.3 MiniCP

MiniCP [2] is a teaching framework for constraint programming implemented in Java. Beacause it is a teaching environment and is smaller than state of the art CP engines, it is not focused on efficiency but rather on readability and thorough testing. All these qualities makes it an ideal framework for comparing different methods for solving CP problems and for testing new CP-oriented data structures. For more details go to [4]. You can also find more documentation on the classes described below in the MiniCP API: [5].

MiniCP has the same key classes and components as most CP frameworks:

- Variable objects that have a stateful domain that can be updated throughout the optimization process, and restored to previous states during backtracks. In MiniCP, there are integer variables (*IntVar*) and boolean variables (*BoolVar*), which are just IntVars with domain $\{0, 1\}$. Their domains are implemented thanks to the *StateInt* and *StateSparseSet* classes. *StateInt* implements a stateful integer, whose value can be saved and restored later. *StateSparseSet* implements a stateful set of integers whose elements can be removed one by one, and restored if needed.

- Constraint classes, which model mathematical or combinatorial constraints on variable objects. All Constraint objects have 2 main methods: *post* and *propagate*, which are used to update the domains of the variables they are bound to. *post* is called only once when the constraint is declared to the solver. *propagate* is called by the solver at every step of the search whenever specific domain listeners detect some change in the key variables. Each constraint has its own way of using its *post* and *propagate* methods.

- A solver class which handles the search and the domain listeners, manages the states of each variable and ties everything together.

2.4 The Insertion Sequence Variable

To model the sequences of places visited by each vehicle in the DARP, we need some kind of reversible data structure that MiniCP can use. The traditional way to do this is with a reversible successor array: for instance we can define
$succ[i]$ = the vertex that comes after i in the sequence, where *succ* is a reversible integer array (StateInt[] in [5]). Fig. 2.3 shows an example of this. Because the routes are mutually exclusive in the DARP, we only need one array to model all of them.

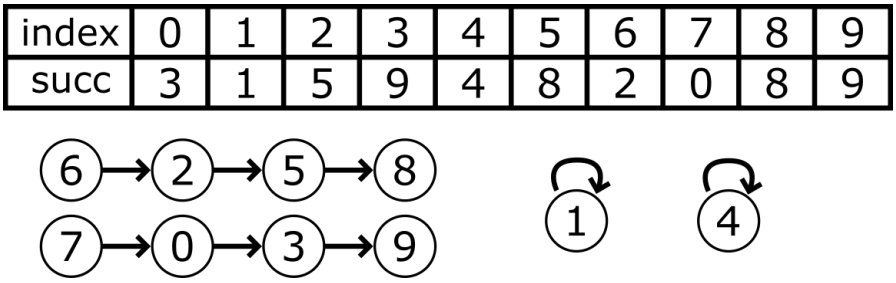

Figure 2.3: Successor array example

Here is how the insertion of new elements in a successor array works. To insert a new element e in a sequence (it does not matter which sequence it is) after p, first $succ[e]$ is set to $s = succ[p]$, then $succ[p]$ is set to e. Fig.2.4 shows the insertion of 1 after 3 in the example. Most of the state of the art DARP solvers use successor

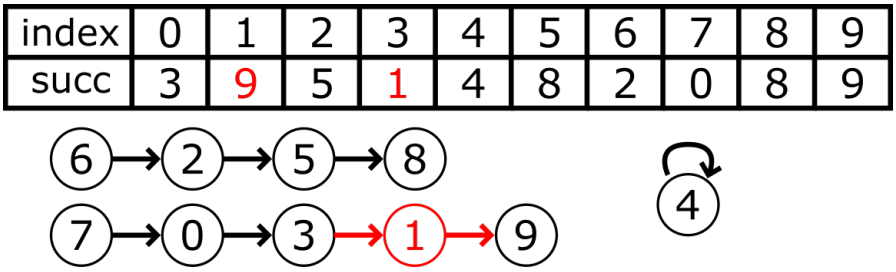

Figure 2.4: Successor array insertion

arrays, and so does the LNS-FFPA solver presented in [1].

In [3], Charles Thomas, R. Kameugne and P. Schaus presented an alternative approach to modeling routing and sequencing problems with a new reversible structure: the Insertion Sequence Variable. In this section we will describe what it consists of, how it can evolve in a CP environment and what constraints can act on it from a theoretical point of view. The implementation of the Insertion Sequence Variable and its constraints for the DARP will be discussed in chapter 3.

An Insertion Sequence Variable or ISV is a more elaborated version of a double reversible successor array. To be more specific, an ISV Sq on an integer set X is a variable whose domain is represented by a tuple $< \vec{S}, R, P, E, I >$ such as:

- $\vec{S}$ is a sequence of elements of X, in a specific order.
- R is the mandatory set: it contains all the elements of X that must be in $\vec{S}$.

- P is the uncertain set: all the elements of X that can be inserted in $\vec{S}$.
- E is the exclusion set: all elements of X that cannot be in $\vec{S}$.
- I is a set of tuples $(e,p)|e \neq p, e \notin \vec{S}, e \notin E$, each corresponding to a possible insertion (insert e after p in $\vec{S}$). p can also take the value $\perp$, which is a special symbol representing the start of the sequence.

R, P an E are mutually exclusive.

Fig. 2.5 shows a graphical representation of the following example ISV:
$X = \{0,1,2,3,4\}$
$Sq =< \vec{S}, R, P, E, I >$
$\vec{S} = \{0,1,3\}$
$R = \{0,1,3\}, P = \{2\}, E = \{4\}$
$I = \{(2,\perp),(2,0),(2,1)\}$

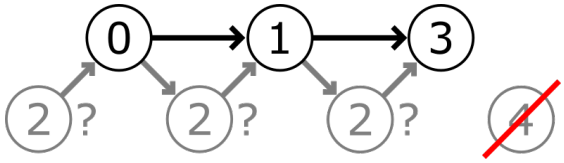

Figure 2.5: ISV example

2.4.1 Operations on ISVs

An ISV has 7 basic methods:

- **isMember(Sq,e)** : return *true* if and only if $e \in \vec{S}$.
- **nextMember(Sq,e)** : return the successor of e in $\vec{S}$.
- **canInsert(Sq,e,p)** : return *true* if and only if $(e,p) \in I$.
- **remInsert(Sq,e,p)** : remove (e,p) from I if $(e,p) \in I$.
- **require(Sq,e)** : add e in R and remove it from P. Fail if $e \in E$.
- **exclude(Sq,e)** : add e in E and remove it from P, then remInsert(Sq,a,b) for all $a = e$ or $b = e$. Fail if $e \in R$.

- **insert(e,p)** : insert e after p in $\vec{S}$, require(Sq,e), then remInsert(Sq,e,x) for all $x \in \{X \cup \perp\}$. Fail if $e \in E$ or if $(e,p) \notin I$.

Note that insertions can only be removed from I, and elements can only be moved from P to E or R, never the other way around. If an ISV Sq has $P = \emptyset$ and $\vec{S}$ contains all the elements of R, then we say that Sq is *bound*.

2.4.2 Constraints on ISVs

[3] Also shows several constraints that can be associated with an ISV. Let's examine the ones that can be useful for our DARP. In all examples below, we assume $Sq =< \vec{S}, R, P, E, I >$.

First and Last

First(Sq, f) ensures that f will always be the first element of $\vec{S}$. It does this by inserting f at the start of $\vec{S}$ and removing any insertion $(e, \perp)$ from I. If f is already in the sequence and is not the first element at the time the constraint is posted, the constraint fails.
Similarly, *Last(Sq, l)* ensures that l will always be the last element in of $\vec{S}$ by removing any insertion (e, l) from I.
Both of these constraints only propagate when they are posted.

Precedence

Precedence(Sq, $\vec{O}$) ensures that the elements in the sequence $\vec{O}$ appear in the same order in $\vec{S}$, if they appear at all. For instance $\vec{S} = \{1,3\}, \vec{O} = \{1,2,3\}$ satisfies the constraint but $\vec{S} = \{3,1\}, \vec{O} = \{1,2,3\}$ does not.
Precedence executes 2 algorithms when it propagates. The first one is executed whenever $\vec{S}$ is modified and simply iterates $\vec{S}$ and $\vec{O}$ conjointly, checking that their common elements appear in the same order. If they don't, the constraint fails.
The second algorithm is triggered when $\vec{S}$ or I is modified, and is used to remove from I all the insertions that would not respect the order of $\vec{O}$. It does this by calling *PrecFiltering* (alg.4) twice: once forwards and once backwards.

Algorithm 4: Algorithm 4: PrecFiltering($Sq, \vec{O}, positions$)

Data: $D(Sq) =< \vec{S}, I, R, P, E >$: Sequence variable domain, $\vec{O}$ = sequence of elements (reversed in second call), $positions[i]$ = index of i in $\vec{S}$.

```
1  precPos ← −1 /* initialized to |S| when reversed */
2  for each(e ∈ O⃗) do
3      if isMember(Sq, e) then
4          precPos ← positions[e];
5      else
6          for each(p ∈ S⃗|(e, p) ∈ I ∧ positions[p] < precPos) do
               /* < changed to ≥ when reversed */
7              remInsert(Sq, e, p);
8          end
9      end
10 end
```

Dependency

Dependency(Sq, U) ensures that either all the elements of the set U are in the sequence, or none of them are. The propagation triggers whenever an element of U is inserted, excluded or required. If it was excluded, then we *exclude(Sq, e)* for all $e \in U$. In the other cases, we *require(Sq, e)* for all $e \in U$.

Cumulative

In the context of the DARP, we can see each request i as an activity A_i composed of the two elements $(start_i, end_i)$ corresponding to the embarking and disembarking, and consuming a load $load_i$. In our case, $start_i = v_i$, $end_i = v_{i+n}$ and $load_i = q_i$.
Cumulative(Sq, [start], [end], [load], C) ensures that the total load never exceeds C at any point in the sequence Sq.
This constraint is propagated whenever a new element is inserted in $\vec{S}$. The propagation filters the insertions from I that are not supported. An insertion for one extremity of an activity is supported if there exists at least one insertion for the other extremity that does not overload the capacity C at any point between the 2 extremities.
This filtering is done in 3 steps:

1. Build the minimum load profile for $\vec{S}$, which indicates the minimum load at each point in the sequence. This is done with one pass over $\vec{S}$. If only one extremity of an activity is present in $\vec{S}$, we build the profile as if the other

extremity was at the closest insertion point. If this assumption leads to a violation of the maximum load C, the constraint fails.

2. Filter the insertions of partially inserted activities. To do this, for each partially inserted activity i, we iterate the insertion points starting from the already present extremity and towards the feasible end (forwards if end_i is missing from the sequence, backwards otherwise) and continue as long as the insertions are supported. As soon as they aren't, the iteration stops and all the remaining insertion points for the missing extremity are removed from I.

3. Use *CumulFiltering* (alg.5) to filter insertions for activities that are not in the sequence at all. The loop at line 5 iterates over $\vec{S}$ from the start ($\perp$). When a potential insertion point for $start_i$ is found, it is added to the *activeStarts* set at line 7. *canClose* indicates if there exists at least one active start position that would be supported if end_i was inserted at the current position. If that is the case and if *current* is a valid insertion point for end_i (line 14), then this insertion point is validated, as well as all the predecessors for $start_i$ in *activeStarts*. All the insertions that are not validated will be removed from I at the end. Whenever the added activity overflows the load profile computed at step 1 (line 10), *activeStarts* is emptied and *canClose* is set to false.

Algorithm 5: Algorithm 5: CumulFiltering($Sq, start, end, load, C, profile$)

Data: $start, end, load$: starts, ends and loads of the activities, C: capacity, $Sq =< \vec{S}, I, R, P, E >$: Sequence variable, $profile$: minimum load profile

```
for each(i|start_i ∉ S ∧ end_i ∉ S) do
    activeStarts ← ∅;
    current ←⊥;
    canClose ← false;
    do
        if canInsert(Sq, start_i, current) then
            activeStarts ← activeStarts ∪ current;
            canClose ← true;
        end
        if profile(current) + load_i > C then
            activeStarts ← ∅;
            canClose ← false;
        end
        if canInsert(Sq, end_i, current) ∧ canClose then
            current is a valid predecessor for end_i;
            ∀p ∈ activeStarts, p is a valid predecessor for start_i;
            activeStarts ← ∅;
        end
        current ← nextMember(Sq, current);
    while current ≠⊥;
    remove predecessors for start_i and end_i that have not beed validated;
end
```

TransitionTimes

TransitionTimes(Sq, [start], [dur], [[trans]]) links the sequenced elements in $\vec{S}$ with their time window to make sure that the transition time inequalities are satisfied between any two consecutive elements of the sequence. More formally, each element $i \in X$ is associated with an activity defined by a start variable $start_i$ and a duration dur_i. Be aware that here, $start_i$ represents a point in time and not a position in the sequence as in the *Cumulative* constraint. A matrix $trans_{i,j}$ satisfying the triangle inequality contains the transition times associated to each pair of activities (i, j). *TransitionTimes* then ensures that

$$start_b \geq start_a + dur_a + trans_{a,b} \tag{2.6}$$

is always true for all pairs $(a \to b)$ of consecutive elements in the sequence.
TransitionTimes propagates whenever a new element is inserted or required in S, or when the bounds of $start_i$ change for any $i \in X$. The constraint propagates in 3 steps:

1. Time window update: run through $\vec{S}$ and update the bounds of the variables in $[start]$ and $[dur]$ so that 2.6 holds true for all pairs of consecutive elements. If the bounds of one variable are shrunk outside of its domain, the constraint fails.

2. Insertion update: run through I and remove any unfeasible insertions according to 2.6. To do this, we have to remove all insertions (e, p) that do not satisfy the following condition:

   $$max(start_p + dur_p + trans_{p,e}, start_e) \leq min(start_s - dur_e - trans_{e,s}, start_e) \tag{2.7}$$

   where s is the successor of p in $\vec{S}$.
   This condition implies 4 simpler conditions by distribution:

   - $start_p + dur_p + trans_{p,e} \leq start_e$, which is equation 2.6 with $(a, b) = (p, e)$
   - $start_e \leq start_s - dur_e - trans_{e,s}$, which is equation 2.6 with $(a, b) = (e, s)$
   - $start_e \leq start_e$, which is trivial
   - $start_p + dur_p + trans_{p,e} \leq start_s - dur_e - trans_{e,s}$, equivalent to $dur_p + trans_{p,e} + dur_e + trans_{e,s} \leq start_s - start_p$. If we put this in the context of the DARP ($start_i = b_i$, $dur_i = d_i$, $trans_{i,j} = dist_{i,j}$), we can see that this is equivalent to $gap_e \geq 0$ (see equation 2.5). In other words, the minimum travel time between p and s through e must be smaller than the maximum amount of time we have between these vertices. This last condition is well hidden but very important, and it is completely independent of $start_e$.

3. Feasible path checking: this is the trickier part. Here, we have to check if there exists a valid extension of the current sequence $\vec{S}$ containing all the current vertices that have been required but not yet inserted. The constraint should fail if there isn't. This problem is NP-Complete by reduction from the TSP. However, we will see in chapter 3 that we do not need to implement this part of the propagation for the purposes of this paper. If you want to read more details on the feasible path checking algorithm, you can find them in [3].

Chapter 3

Implementation

In this chapter we will discuss the details of the implementation of 2 DARP solvers in MiniCP: the first one simply implements the LNS-FFPA search algorithm in MiniCP and serves as a control group. The other one makes use of Insertion Sequence Variables instead of the classical successor array. The code for these implementations are publicly available at my github: [6].
In this chapter, the classes referenced in *italic* are from MiniCP and more documentation about them can be found in the MiniCP API [5] .

3.1 Architecture

Both solvers are integrated inside the MiniCP architecture, which is divided in several main folders. Here are the main ones:

- **src/main/java/minicp/engine** contains the main elements of MiniCP, spread into 2 folders: **core** contains all the classes and interfaces that implement the variables and the MiniCP solver. **constraints** contains all the constraint implementations for said variables.
- **src/main/java/minicp/state** contains the stateful tools and classes that the backtracking mechanisms rely on.
- In **src/main/java/minicp/search** you can find the classes and tools used by the search engine when solving CP problems.
- Last but not least, **src/main/java/minicp/examples** contains all sorts of algorithms that use MiniCP to model and solve various CP problems. This is where our 2 DARP solvers lie, respectively in the **DARP_RK** and **DARP_LNSFFPA_Seqvar** files. They also use **DARPDataModel**, which contains classes that help model various things in the DARP, and

DARPParser, which is used to read the DARP instances at **Data/DARP**. More on these instances in chapter 4.

3.2 Solver A: DARP without ISVs

This solver implements LNS-FFPA in a classical way, using a successor array to model the sequences. Although some simplifications have been made, it is largely inspired by the work of Roger Kameugne and Charles Thomas, who have implemented a similar solver in scala using another cp engine called *Oscar* ([7]).

3.2.1 Variables

Here are the important variables for this solver:

- **currentSolution** is a global pointer to the current solution in LNS-FFPA and **currentSolutionObjective** is its cost.
- **bestSolution** is a global pointer to the best solution found so far, and **bestSolutionObjective** is its cost.
- **succ** is the successor array representing the sequences, just like described in section 2.4. It is implemented as a *StateInt[]* ([5]), i.e. an array of reversible integers. Its size is $2n + 2m$. This is the main variable for our problem. It is accompanied by **pred**, which models exactly the same sequences but in reverse. **pred** is not essential, it just helps with the implementation.
- **servingTime** is an array of *Intvar* of size $2n+2m$ which contains the serving times of each vertex. **servingTime[i]** corresponds to the variable b_i described in section 1.1.3.
- **customersLeft** is a *StateSparseSet* of size n representing the set of all unassigned requests. It is used to apply the heuristics described in section 2.2, and is updated at every new insertion.
- **insertionObjChange** is an array of HashMaps, one for each request, that is used to store the precomputed insertion points for that request and their costs. The details of this are explained below.
- **capacityLeftInRoute** is a *StateInt[]* of size $2n + 2m$ which is used to build a dynamic capacity profile of the routes. More on this in the **Constraints** section.

3.2.2 LNS-FFPA

Let us explore how LNS-FFPA actually works with MiniCP. More accurately, let's see how each of the 3 algorithms of section 2.2 are implemented.

Algorithm 1: TreeSearch

MiniCP plays a big role in this part of the program because **TreeSearch** is a depth-first algorithm that can be supported by MiniCP's search engine. Here is how: in MiniCP, a *DFSearch* object can be declared with a procedure as argument. This procedure takes no argument and will be executed at the start of each node in the depth-first search tree. It must return a group of procedures, which represent the different branches the search can explore.
The **TreeSearch** procedure looks like this:

Listing 3.1: The TreeSearch procedure

```
1   if (customersLeft.isEmpty()) return EMPTY;
2   else {
3       computeInsertionPoints();
4       int request = getUnassignedRequest();
5       int[][] points = getInsertionPoints(request);
6       Procedure[] branches = new Procedure[points.length];
7       for (int i = 0; i < points.length; i++) {
8           int[] p = points[i];
9           branches[i] = () -> branchRequestPoint(request, p);
10      }
11      return branch(branches);
12  }
```

As a first step, **computeInsertionPoints()** (line 3) does exactly what its name suggests: it looks for all feasible insertion points for all the unassigned requests and stores these points in **insertionObjChange**. Consider this function as a black box for now.

Next, **getUnassignedRequest()** is used to choose which request the search will branch on. The procedure will then create one branch for each of the insertion points of the chosen request and return all the branches (lines 5-11). Note that **getInsertionPoints()** retrieves the points from **insertionObjChange** and sorts them in increasing order of cost e, thus the branches will be explored in the same order. **branchRequestPoint(r, p)** is a procedure that will try to insert the request r in the solution at the specified point p and trigger a failure if that insertion violates any of the constraints.

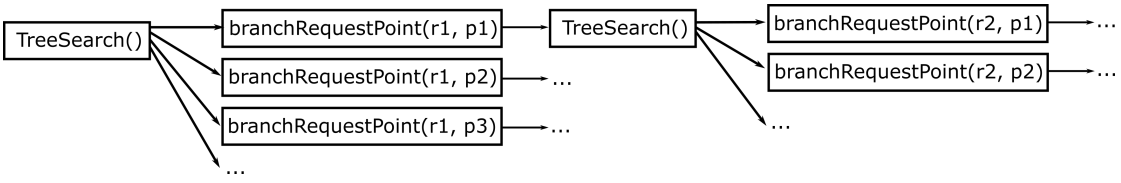

Figure 3.1: branching sheme

Reminder from section 2.2.1: an insertion point for a request r is a tuple $< v, p, d, e >$. Inserting r at this point means inserting v_r after v_p and v_{r+n} after v_d in the route of the vehicle v. e is the insertion cost. In practice, the insertion points are declared as arrays of integers containing these 4 values.

If there are no more unassigned requests (line 1) it means that the solution is complete, so we return *EMPTY* to signal the search engine that we have reached max depth and there are no more nodes to be explored in this branch. Note that Algorithm 1 tells us to stop the search at this point, but remember this is just the declaration of the *DFSearch* our solver is going to use. The way it is used will be described later.

Now let's see how the insertion points are actually computed. Every time **computeInsertionPoints()** is called, **insertionObjChange** is first emptied of all previous points since some of them may not be valid in the new sequence. Then we call the procedure **setInsertionCost(r, v)** for every unassigned request r and every vehicle v.
setInsertionCost(r, v) explores all the possible insertions of r in the current route of vehicle v, and stores all the feasible insertion points found and their cost in **insertionObjChange**. The way it does it is with a double loop:

Listing 3.2: setInsertionCost

```
1  void setInsertionCost(int request, int v) {
2      int begin = getBeginDepot(v);
3      int end = getEndDepot(v);
4      int pPred = begin;
5      while (pPred != end) {
6          if (pPred is a valid predecessor for pickup) {
7              int dPred = succ[pPred].value();
8              while (dPred != end) {
9                  if (dPred is a valid predecessor for drop) {
10                     insertionObjChange[request].add(
11                     {v, pPred, dPred, e});
12                 }
13                 dPred = succ[dPred].value();
14             }
15         }
16         pPred = succ[pPred].value();
17     }
18 }
```

The first loop explores potential predecessors for the pickup vertex ($pPred$) starting at the beginning of the route, and the second tries positions for the delivery vertex ($dPred$) starting at the position after $pPred$. If both positions are valid, then the insertion cost e is computed and the new insertion point added to **insertionObjChange** (lines 10-11). This pseudo-code example is highly simplified. For instance, it does not account for the special case where the drop is right after the pickup, whereas the actual implementation does.
This is where one of the differences with the implementation by Roger Kameugne lies. In his version as well as in the Comet implementation from [1], The first loop explores positions for the critical vertex and the second one for the non-critical vertex. If the critical vertex is the delivery vertex, then the loops go backwards. This makes this part of the solver more complex but also more efficient at finding insertion points.

The computation of the insertion points is a key part of our solver because this is where we try to simulate the insertion of some vertices in the sequences. This means that we can already check the feasibility of an insertion point with respect to some of the constraints here, before the insertion actually happens. In many cases, this pre-propagation of the constraints is more efficient than waiting for the request to actually be inserted and letting MiniCP propagate the constraints. The reasoning behind this is that the sooner we can catch an infeasible insertion, the

easier it is to discard it.
As a result of this, some of the constraints are checked during the computation of the insertion points, and some are checked during the insertions themselves. The details of how each constraint is handled will be covered in the **Constraints** section below.

Algorithm 2: getUnassignedRequest

The implementation of this one is much simpler. As described in alg.2, the goal is to find an unassigned request that:

1. Can be inserted in the fewest routes
2. Has the fewest insertion points
3. Has the cheapest insertion point in terms of e

In practice we start with a list containing of all the requests in **customersLeft**, then filter it 3 times with respect to the 3 criteria in the correct order. Of course we use the precomputed insertion points that were stored in **insertionObjChange** beforehand.

Algorithm 3: MinimizeRoutingCost

The code for the main search algorithm is split into 3 parts. First, the solution handling. *DFSearch* objects have a specific listener for solution finding in the form of the *onSolution()* method. This method is passively called whenever a branch in the search tree is completed, i.e. whenever **TreeSearch** finds a complete feasible solution. This is how we use it:

Listing 3.3: solution handling

```
1  DFSearch search = new DFSearch(TreeSearch);
2  search.onSolution(() -> {
3      double rand = Math.random();
4      int obj = getDistanceObjective();
5      if (obj < currentSolutionObjective || rand < d) {
6          currentSolution = exportSol(obj);
7          currentSolutionObjective = obj;
8          if (currentSolutionObjective < bestSolutionObjective) {
9              bestSolution = currentSolution;
10             bestSolutionObjective = currentSolutionObjective;
11         }
```

```
12        }
13    });
```

At line 4, **getDistanceObjective** computes the cost of the last solution found. This is the actual cost of the solution, not the cost metric e used in algorithms 1 and 2.
At line 6, **exportSol()** retrieves the last solution found (which, at this moment, is only present in the **succ** and **servingTime** variables which will be restored to anterior values).

Next, we have the main search loop:

Listing 3.4: lns

```
1  static void lns() {
2    int i = 2;
3    while (remainTime > 0 && i <= maxSize - range) {
4      if (i==maxSize - range) i=2; // reset loop if there is time left
5      int j = 0;
6      while (remainTime > 0 && j <= range) {
7        int k = 1;
8        while (remainTime > 0 && k <= numIter) {
9          long t1 = System.currentTimeMillis();
10         SearchStatistics stats = search.solveSubjectTo(
11             statistics -> statistics.numberOfSolutions() == 1,
12             () -> {
13                 relax(i + j);
14             });
15         k++;
16         remainTime -= System.currentTimeMillis() - t1;
17       }
18       j++;
19     }
20     i++;
21   }
22 }
```

This the exact same loop system as described in alg.3. At lines 10-11, we call the *solveSubjectTo()* method of *DFSearch*, which has 2 arguments. The first one is a predicate. The search will stop as soon as this predicate is true. In our case, we use it to stop the search at the first solution found. The second argument is a procedure that will be executed before the search starts. Here, we pass on our third element: the **relax** method.

relax(x) does 2 things:

1. select x requests at random among all requests
2. update all the variables so that the current solution state becomes a copy of the solution stored in **currentSolution**, but the x requests selected at step 1 have been removed from the routes. This creates a new partial solution that can be used as a starting point for the solver.

3.2.3 Constraints

In this section, we discuss how each constraint of the DARP is handled in practice, and how this handling fits within the implementation of LNS-FFPA. Reminder: you can find the list of all the DARP constraints in section 2.1.

Time Window

To satisfy the time windows, we simply initialize the domain of **servingTime[j]** as $[e_j, l_j]$ for every vertex j.

First & Last

As for the time windows, this one is handled during the initialization of the variables. For every vehicle v, **succ[2n+v]** is set to $2n + m + v$, defining the initial route as $\vec{R_v} = \{startDepot_v, endDepot_v\}$.
Furthermore, due to the bounds of the loops in **setInsertionCost**, any insertion of a vertex before a start depot or after an end depot will never be considered.

Maximum Ride Time

Here we can use MiniCP's *LessOrEqual* constraint. For every request i, we post $LessOrEqual(\textbf{servingTime[n+i]}, \textbf{servingTime[i]} + d_i + L)$.

Maximum Route Duration

For every vehicle v, the constraint
$LessOrEqual(\textbf{servingTime[2n+m+v]}, \textbf{servingTime[2n+v]} + D)$ is posted.

Dependency

For this one, there actually is nothing more to be done. Since the insertion points are defined by request, there is no way for a pickup vertex to be inserted in a route without its corresponding delivery vertex and vice versa. **setInsertionCost** will always consider the insertions conjointly. Hence **Dependency** is already taken care of by the branching scheme of LNS-FFPA.

Precedence

As for **Dependency**, the branching scheme and the nature of the insertion points passively take care of **Precedence**. **setInsertionCost** will never attempt to insert the vertices of a request in the wrong order.

Cumulative

To ensure this constraint is always satisfied, we use a similar technique as described in section 2.4.2. The global variable **capacityLeftInRoute** is used to maintain a minimum load profile for all the routes. At any given moment, **capacityLeftInRoute[i]** indicates the remaining capacity in the vehicle after visiting vertex i. The load profile of a route v is updated whenever a new request is inserted in it. To do this we call **updateCapacityLeftInRoute(v,p)**, where p is the predecessor of the newly inserted pickup vertex.

Listing 3.5: load profile update

```
1  void updateCapacityLeftInRoute(int v, int p) {
2      int end = getEndDepot(v);
3      int index = p;
4      int capacity = capacityLeftInRoute[index].value();
5      while (index != end) {
6          capacity = capacity - q[index];
7          capacityLeftInRoute[index].setValue(capacity);
8          index = succ[index].value();
9      }
10 }
```

This procedure does one pass through the modified route, starting at p and ending at the end depot $2n + m + v$. The capacity left is updated by subtracting the load q_{index} of each stop along the way. Remember that pickup stops have a positive load and delivery stops have a negative load, so they respectively make us lose or gain space in the vehicle. Fig. 3.2 illustrates what the update does in practice.

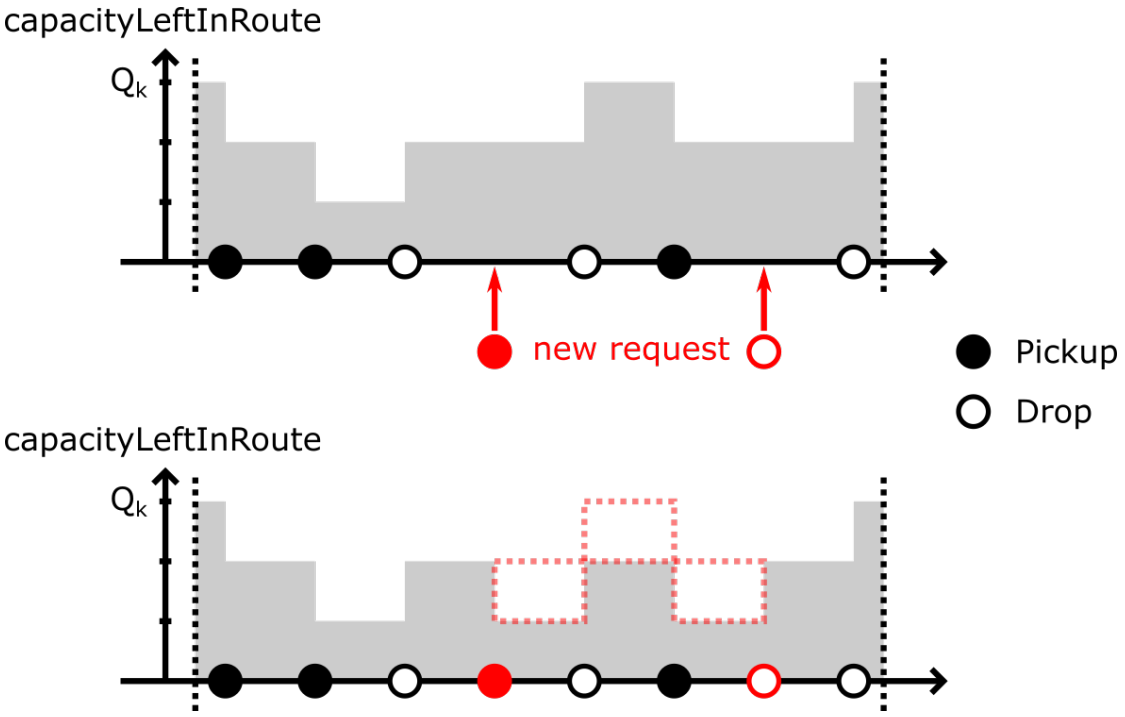

Figure 3.2: load profile update

But how do we use this load profile to filter out bad insertions? In the last section, when discussing the implementation of **TreeSearch**, it was mentioned that constraints could be checked either after the insertions in **branchRequestPoint** or during the insertion point computing in **setInsertionCost**. For the **cumulative** constraint, a complete and exact pre-propagation can be done by modifying the loop structure of **setInsertionCost** a little bit:

Listing 3.6: setInsertionCost with capacity check

```
1  void setInsertionCost(int request, int v) {
2      int begin = getBeginDepot(v);
3      int end = getEndDepot(v);
4      int pPred = begin;
5      while (pPred != end) {
6          if (pPred is a valid predecessor for pickup
7                  && capacityLeftInRoute[pPred].value() >= q[pickup]) {
8              int dPred = succ[pPred].value();
9              while (dPred != end
10                     && capacityLeftInRoute[dPred].value() >= q[pickup]) {
11                 if (dPred is a valid predecessor for drop) {
12                     insertionObjChange[request].add(
```

```
13                          {v, pPred, dPred, e});
14                      }
15                      dPred = succ[dPred].value();
16                  }
17              }
18              pPred = succ[pPred].value();
19          }
20  }
```

At line 7, we add a new condition for $pPred$ to be a valid successor: there must be enough space left in the vehicle at that position in the sequence. At line 10, the same condition is checked for $dPred$, but in the loop this time. This is because we are looking for a sub-sequence in the route that can support our request along its whole length. As soon as this condition is false, it means that the last value found for $pPred$ can no longer be supported, so it is useless to look further in the $dPred$ loop.

Transition Times

The **Maximum Ride Time** and **Maximum Route Duration** are simple to implement, but they rely on a consistent and regular update of the **servingTime** variables. This is where **Transition Times** comes into play.
In this approach to the DARP, the **Transition Times** constraint has 2 jobs to do.

Job 1: Whenever a new vertex is added to a route, we must post new constraints enforcing that $b_j \geq b_i + d_i + dist_{i,j}$ stays true for all pairs (i, j) of consecutive stops in the route. This condition must be preserved even after the insertion, because future insertions will keep updating the serving times.
To achieve this, in the **branchRequestPoint** procedure, when a new vertex j is inserted in the route between its predecessor p and its successor s, we post $LessOrEqual(\textbf{servingTime[p]}) + d_p + dist_{p,j}, \textbf{servingTime[j]})$ and $LessOrEqual(\textbf{servingTime[j]}) + d_j + dist_{j,s}, \textbf{servingTime[s]})$.

With all these inequality constraints accumulated, every new insertion triggers a chain propagation updating the bounds of **servingTime** for all the stops in the route, including the newly inserted vertex.

Job 2: Knowing the transition time conditions in advance, we can also check if an insertion is feasible before it happens. To do this, we can borrow equation 2.7 from section 2.4.2 and apply it to our case.

$$max(b_p + d_p + dist_{p,e}, b_e) \leq min(b_s - d_e - dist_{e,s}, b_e) \tag{3.1}$$

This gives us a condition that any insertion of a vertex e between p and s must satisfy. If it doesn't, then p is not a valid predecessor for e. In the implementation, **setInsertionCost** uses this equation to check the validity of the pickup and drop predecessors.

Summary

This table (3.1) briefly recapitulates how and when each of the constraints is handled in solver A. I stands for "posted once at initialization", S stands for "explicitly

Constraint	I	S	B	P
Time Window	x			
First & Last	x			
Dependency				x
Precedence				x
Max Ride Time	x			
Max Route Duration	x			
Cumulative		x		
Transition Times		x		
Inequality constraints for Trans. Times			x	

Table 3.1: Constraint handling in solver A

checked when searching for insertion points", B means "posted during branching" and P means "passively handled".

3.3 ISV implementation

Before diving into the implementation of the ISV solver, let's first cover how the ISV itself is implemented in MiniCP. This section describes a MiniCP adaptation of the ISV implementation described in [3]. The code for the ISV follows the same architecture as the *Intvar*: all ISV objects are accessed through the **InsertionSequenceVar** interface. The main class implementing this interface is **ISVImpl**, but the domain of the ISV is implemented in a seperate class called **ISVDomain**. These 3 files can be found in **minicp/engine/core**. Each of the ISV-related constraints has its own class in **minicp/engine/constraints**.

ISV domain

As said in section 2.4, the domain of an ISV defined over a set X of size n is a tuple $< \vec{S}, R, P, E, I >$. Here is how each part is implemented:

- The sequence $\vec{S}$ is modeled by 2 successor arrays **succ** and **pred**, which are *StateInt[]*, just like in the external implementation of the sequences. In this context however, their size is $n+1$ because there is an additional slot for the start symbol $\perp$.
- The exclusion set E, the mandatory set R and the uncertain set P are mutually exclusive, so they can implemented using a single array of size n called **elems** and two *StateInt* **r** and **p**, whose values are indexes that respectively separate R from P and P from E, as illustrated in fig.3.3. Along these is **elemPos**, another array of size n. **elemPos**[e] indicates the index of the element e in **elems**. Hence, we can easily tell which of the 3 sets contains any given element e by comparing **elemPos**[e] to **p** and **r**.
- I is implemented using a *StateSparseSet[]* of size n named **posPreds**. Each of these *StateSparseSet* is of size $n+1$ and contains the potential predecessors of the corresponding element. For instance, the **posPreds** example shown in fig.3.3 corresponds to $I = \{(c, \perp), (c, e), (c, f), (e, c), (e, f)\}$. By default, each element e is removed from **posPreds**[e] because e cannot be inserted after itself.

Figure 3.3 is taken from [3].

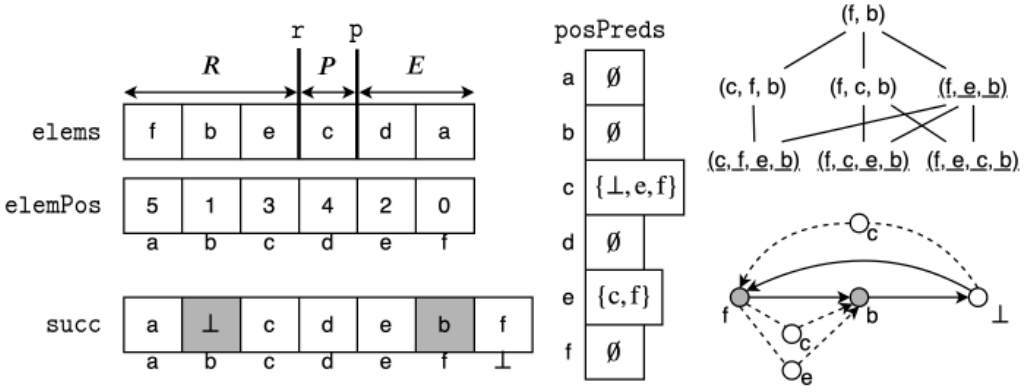

Figure 3.3: ISV domain implementation

ISV main methods

Here are the main methods of *InsertionSequenceVar*:

- **isMember(e)** returns *true* iif e is part of the sequence $\vec{S}$.
- **nextMember(e)** returns the successor of e in the sequence.
- **prevMember(e)** returns the predecessor of e in the sequence. These methods let us iterate the sequence both ways easily.
- **remInsert(e,p)** simply removes p from **posPreds**[e].
- **canInsert(e,p)** returns *true* if **posPreds**[e] contains p, false otherwise.
- **exclude(e)** first moves e from P to E. This is done in 2 steps:
  1. e is swapped with **elems**[**p-1**], and so are their positions in **elemPos**.
  2. the value of p is decremented.

  Next, **posPreds**[e] is emptied and **remInsert(i,e)**] is called for all $i \in X$. **exclude(e)** fails if e is already in R, i.e. if **elemPos**[e]<**r**.
- **require(e)** does a similar swapping operation, but on the other side of the domain to move e from P to R:
  1. e is swapped with **elems**[**r**], and so are their positions in **elemPos**.
  2. the value of r is incremented.

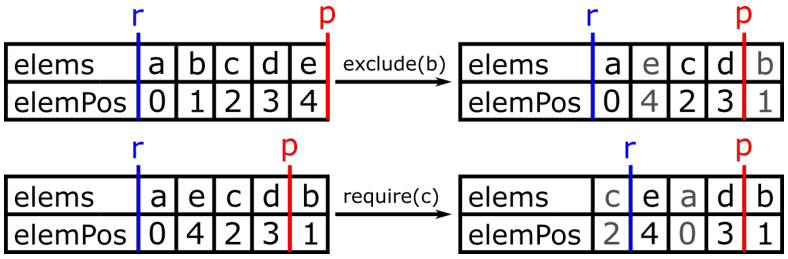

Figure 3.4: exclude & require

This "swap and increment" technique is also what is used to implement the *StateSparseSet* in MiniCP, only that we need two separation indices for the ISV domain and *StateSparseSet* only uses one, which makes it impractical to use in our case. **require(e)** fails if e is already in E, i.e. if **elemPos**[e] $\geq$ **p**.

- **insert(e,p)** does 3 things: insert e after p in **succ** and **pred**, **require(e)** and empty **posPreds**[e]. **insert(e,p)** fails if **canInsert(e,p)** is $false$.

ISV constraints

The ISV has several dedicated constraints, but not all of them need to be implemented in their own class for this DARP solver to work. The goal of this implementation is to solve a DARP using ISVs, not to implement all the constraints that ISVs can use. Thus each DARP constraint is implemented in a manner that fits this solver best, the details of which are covered in the **Constraints** section below. Nevertheless, here are the ISV constraints mentioned in section 2.4.2 that were implemented for this solver:

- **First** is very easy to implement with ISVs. Its *propagate()* method only executes when **First(V,f)** is posted, so it doesn't need any domain listener like most MiniCP constraints do. V is the *InsertionSequenceVar* **First** applies to and f is the element that should be at the start of the sequence. The propagation does 2 things:
  - **V.insert(f,⊥)**
  - **V.remInsert(i,⊥)** for all $i \in X$.
- **Last(V,l)** works the exact same way but instead does:
  - **V.insert(l,j)** j being the current last element in the sequence.
  - **V.remInsert(i,l)** for all $i \in X$.
- **TransitionTimes** is by far the most useful ISV constraint for the DARP. Its constructor is defined as **TransitionTimes(InsertionSequenceVar V, IntVar[] start, int[] dur, int[][] tt)**. It propagates whenever an element is inserted in V, or whenever the bounds of one of the variables in *start* change. Its propagation algorithms are implemented with the DARP in mind.

  1. **timeWindowUpdate()** 3.7 iterates through V, posting the adequate *LessOrEqual* constraint to enforce equation 2.6 between each pair of consecutive elements. If there are less than 2 elements in V (line 2), there is nothing to do. Notice that the constraint deactivates itself between lines 3 and 12. This is because during that time the bounds of the *start* variables will be updated, but we don't want these updates to trigger **TransitionTimes** again. Deactivating the constraint during that time prevents it from being scheduled for propagation too many times.

Listing 3.7: time window update

```
1 void timeWindowUpdate() {
```

```
2       if (V.size() < 2) return;
3       this.setActive(false);
4       int last = V.nextMember(⊥);
5       int current = V.nextMember(last);
6       while (current != ⊥) {
7           cp.post(lessOrEqual(start[last],
8           dur[last] + tt[last][current], start[current]));
9           last = current;
10          current = V.nextMember(current);
11      }
12      this.setActive(true);
13  }
```

2. **InsertionUpdate()** checks equation 3.1 for all the insertions (e, p) where **isMember(e)** $= false$ and **isMember(p)** $= true$. The successor of p is **nextMember(p) = s**. If
   $max(start[p].min() + dur[p] + tt[p][e], start[e].min()) >$
   $min(start[s].max() - dur[e] - tt[e][s], start[e].max())$
   then we call **V.remInsert(e,p)**.
3. **feasiblePathChecking()** normally is an important part of the propagation of **TransitionTimes**, however it only is useful if there are elements ins V that have been required but not inserted yet, and this never happens in this approach to the DARP since the requests are never partially inserted.

3.4 Solver B: DARP with ISVs

This solver is the main contribution of this paper. It is largely based on solver A, the main difference being that this time, the routes of the DARP are modeled by insertion sequence variables instead of a successor array. To be more specific, the **succ** and **pred** variable have disappeared. Each vehicle v now has its own sequence modeled by an ISV called **route[v]**.

3.4.1 LNS-FFPA

LNS-FFPA is implemented the same way as in solver A, and has the same key methods although some of them have small modifications. The same HashMap system for pre-computing, storing and fetching the insertion points is used. Here are the key changes in the main methods:

setInsertionCost

The only change in this method is the conditions used to check if a drop/pickup predecessor is valid. More on this in the **Constraints** section below.

branchRequestPoint

The difference here lies in the insertion of the requests. In solver A the vertices would be inserted directly by updating **succ** and **pred** and inequality constraints had to be posted at the same time to update **servingTime**. In solver B, we simply call **route[v].insert(pickup, p)** and **route[v].insert(drop, d)** to apply an insertion point $< v, p, d, c >$. These calls essentially accomplish the same things but internally.

3.4.2 Constraints

This section describes how each constraint of the DARP is implemented in this solver.

First & Last

The **First** and **Last** constraints of the ISV are obviously made for this job. We simply have to post **First(route[v], 2n+v)** and **Last(route[v], 2n+m+v)** for each vehicle v.

Dependency & Precedence

Just like in solver A, the insertions are grouped by request so the 2 vertices of a request will always be inserted together and in the right order. These constraints are handled by the search structure itself.

Time Window, Max Ride Time & Max Route Duration

These are handled the exact same way as in solver A.

Cumulative

A specific **Cumulative** constraint for the ISV could have been used here, however the system used in solver A is already very efficient and achieves the same filtering of the insertion points. This solver thus uses the same system which relies on the load profile stored **capaciyLeftInRoute**. Like in solver A, **capaciyLeftInRoute** is updated after each new insertion in **branchRequestPoint** and used to check the validity of insertion points in **setInsertionCost**.

TransitionTimes

It is this constraint that generates the largest differences in the codes of solvers A and B. While the first solver had to explicitly post inequality constraints at each new insertion in **branchRequestPoint** and check equation 3.1 for each predecessor in **setInsertionCost**, in the second solver we just have to post
TransitionTimes(route[v], servingTime, d, dist)
for each vehicle (d is an array of size $2n + 2m$ containing the serving duration of each vertex) and all these things will be done passively. **timeWindowUpdate** will post the adequate inequality constraints at each insertion and **insertionUpdate** will take care of checking equation 3.1 and filtering out the invalid insertions in the routes. This way, **setInsertionCost** only has to call **route[v].canInsert(e,p)** to know if **TransitionTimes** allows us to insert e after p in **route[v]** or not, instead of explicitly checking equation 3.1.

Summary

This table (3.2) briefly recapitulates how and when each of the constraints is handled in solver B. I stands for "posted once at initialization", S stands for "explicitly

Constraint	I	S	B	P
Time Window	x			
First & Last	x			
Dependency				x
Precedence				x
Max Ride Time	x			
Max Route Duration	x			
Cumulative		x		
Transition Times	x			
Inequality constraints for Trans. Times			x	

Table 3.2: Constraint handling in solver B

checked when searching for insertion points", B means "posted during branching" and P means "passively handled".

Chapter 4

Experimental results

This chapter presents several experiments testing the solvers described above. The DARP instances used in these experiments are were given by R.Kameugne's implementation of LNS-FFPA. They are based on the instances from Cordeau et al. [8] which are based on realistic assumptions and data provided by the Montreal Transit Commission (MTC). Half of the requests are outbound and half inbound. They are divided into classes a and b, the difference being that class a instances have tighter time windows. The instances are noted as [class][m]-[n], m being the number of vehicles and n the number of requests.
All the experiments were run on the same computer: a AMD Ryzen 5 3600 processor with 16 GB of RAM.

By default, the parameters for LNS-FFPA in both solvers are $\alpha = 1$, $\beta = 0$, $range = 4$, $numIter = 300$, $d = 0.07$ and $maxSize = n/2$. They are the same LNS-FFPA parameter values as in [1], except for α and β. More on this in experiment 3.

4.1 Experiment 1: Optimality

This first experiment tries to determine if the 2 solvers will always find the optimal solution of a DARP instance given enough time.

4.1.1 setup

To do this, we need to compare them with a third solver that we know is optimal. This solver is simply a brute force approach: instead of solving for the first feasible solution and using it as a starting point for LNS-FFPA, we directly solve for all

solutions an return the best one. This approach uses the same search heuristic as the other solvers, but we know this one will search the whole solution space.

The brute force solver is very slow, so we need to run this test on DARP instances that are small enough to be solvable by brute force in a reasonable amount of time, but large enough that the optimal solution is not trivial. The instances a2-16 and b2-16 fit this description.
The results show the cost of the best solution, the total number of failures, and the time taken by each solver to find its best solution.

4.1.2 results & discussion

A single run with the default parameters gives these results (tab.4.1):

Instance		Brute force	noISV	ISV
a2-16	cost	294.26*	294.26*	294.26*
	fails	70144	14	72
	time (s)	117	<1	2
b2-16	cost	309.39*	309.61	309.61
	fails	57475	1	3
	time (s)	235	<1	3

Table 4.1: optimality

We can see that we indeed reach optimality with a2-16 but not with b2-16. Both solver A and solver B kept searching indefinitely without finding the optimal solution with cost 309.39. Why is that? We came very close, so why couldn't LNS-FFPA find the optimal solution in a reasonable amount of time? The answer lies in 2 other observations that are not in this table.
First, the best solution (309.39) was found very late by the brute force solver while the second best (309.61) was found very quickly. This means that the heuristic guides the search towards the latter very well but not towards the former, which is the one we are looking for.
Secondly, Even though the best solution and the second best are very close in terms of cost, their routes are very different, indicating that they were not in a sufficiently close neighbourhood in the search space. This could explain why LNS-FFPA could not likely find one starting from the other.

This example does not prove that LNS-FFPA is not an optimal search algorithm however. We should be able to find the best solution of instance b2-16 with LNS-FFPA if we tune the parameters to widen the search (tab.4.2). Let's use

$range = 5$ instead of 4 and $d = 0.3$ instead of 0.07, meaning that we explore a larger neighborhood with a higher probability of accepting solutions with a higher cost.

Instance		Brute force	noISV, widened	ISV, widened
a2-16	cost	294.26*	294.26*	294.26*
	fails	70144	17	4846
	time (s)	117	<1	75
b2-16	cost	309.39*	309.39*	309.39*
	fails	57475	310	484
	time (s)	235	63	80

Table 4.2: optimality

The widened search did in fact find the best solution for b2-16, confirming that the default LNS-FFPA was stuck in a distant region of the solution space, too far away from the optimal solution. It was also searching in a small neighborhood, making it very unlikely to find the optimal solution quickly.
The modified LNS-FFPA was also slower at finding the optimal solution for a2-16, indicating that the default parameters were better suited for that instance.
Keep in mind that these results were based on a single run, so the time values may not accurately reflect what happens on average. That said, producing decent means would take a lot more time, and the current results are meaningful enough for the purpose of this experiment.

In conclusion, we can say with fair confidence that LNS-FFPA is an optimal search algorithm. The catch is that some DARP instances are harder than others, and the search parameters must be tuned accordingly for optimal performance.
This is not a big surprise. After all, given unlimited time, LNS-FFPA is more or less an endless loop that randomly explores the neighbourhood of the current solution, using a heuristic whose influence can be adjusted by the search parameters. This algorithm is bound to find the optimal solution at some point, even if it takes a very long time.

4.2 Experiment 2: Convergence

The goal of this experiment is to show how the cost of the best solution found evolves as the search progresses. [1] showed that LNS-FFPA could find good solutions very quickly. We will try to observe that this test.

4.2.1 setup

To perform this experiment, we just need to solve some instances with one of our solvers and plot the cost of the best solution against the elapsed time as the solver works. Solver B was used for this task, with default search parameters.

4.2.2 results & discussion

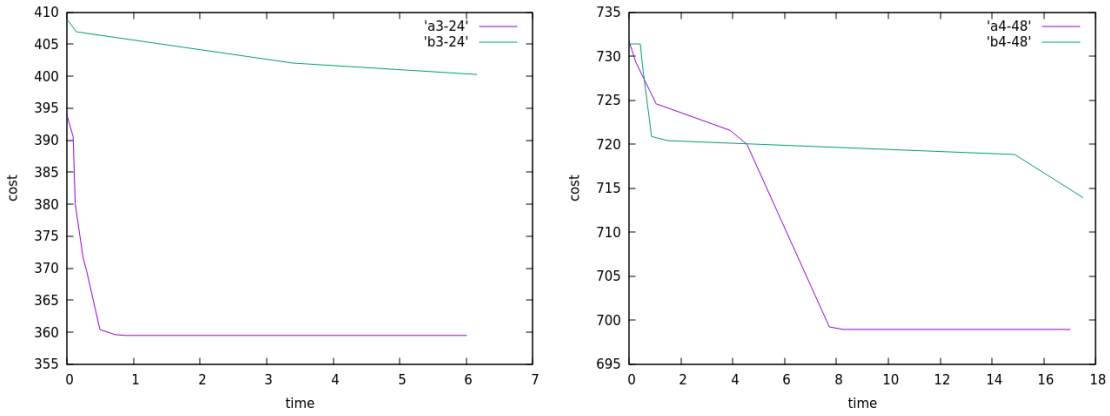

Figure 4.1: Cost of best solution against time, small instances

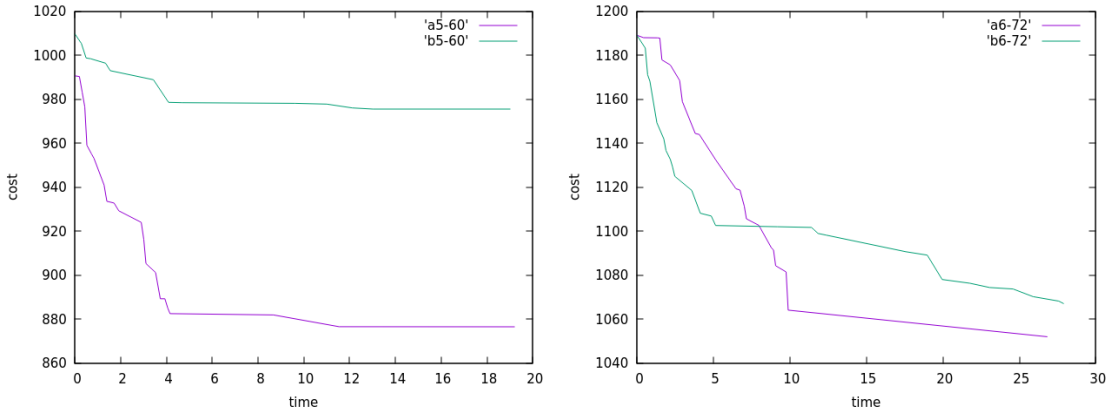

Figure 4.2: Cost of best solution against time, medium instances

While the small instances (fig.4.1) do not give enough data points to show anything relevant, on the medium and large instances (fig.4.2 and 4.3) we can observe an asymptotic convergence that looks like an exponential distribution, which is exactly what is expected of the solution quality over time in a DARP solver.

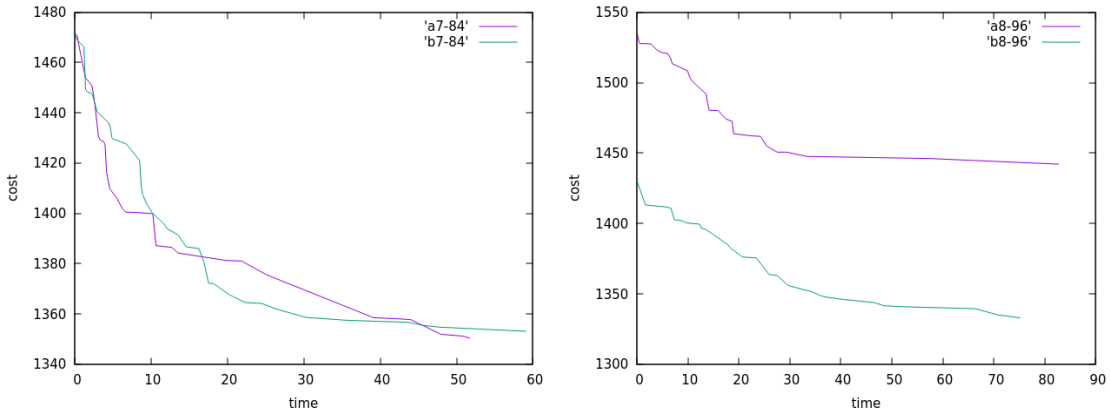

Figure 4.3: Cost of best solution against time, large instances

4.3 Experiment 3: Slack

In this test, we will try to determine the optimal values for the meta-parameters α and β described in section 1.2.1. Due to the usage of these parameters in equation 2.4, it is the slack ratio $\frac{\alpha}{\beta}$ rather than the actual values of α and β that is most important. In [1], the values $\alpha = 80$ and $\beta = 1$ were used. This experiment will tell us if these values still apply to our solvers to get good results.

4.3.1 setup

The influence of the α and β parameters fully lies in the search heuristic and its cost metric e. From this we can make 2 assumptions: first, it does not matter which solver (A or B) we use since both use the exact same search heuristic. Secondly, only the branching choices are affected by the values of α and β and LNS-FFPA itself has nothing to do with it. This means that we can limit the search to the initial solution found by **TreeSearch**. We can thus save a lot of time and plot the cost of the initial solution against $\frac{\alpha}{\beta}$ for lots of values and instances.

4.3.2 results & discussion

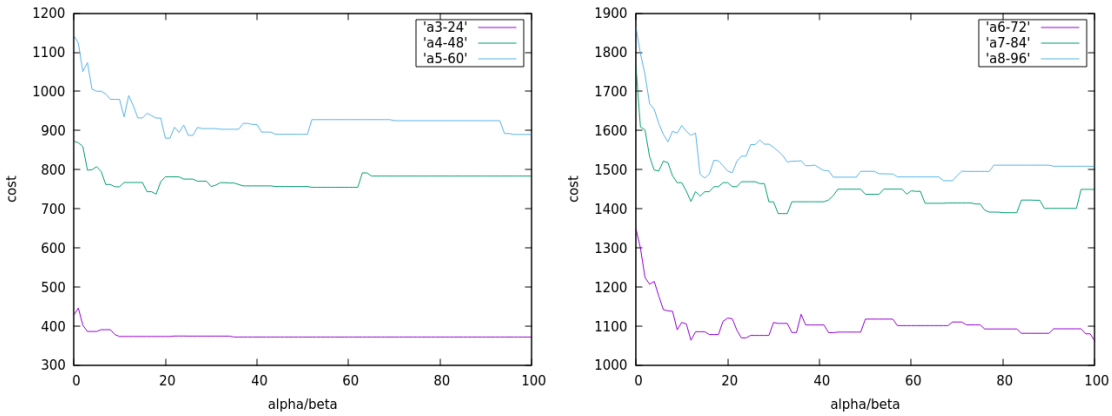

Figure 4.4: Cost of initial solution against $\frac{\alpha}{\beta}$, class a

Let's first try small values $\frac{\alpha}{\beta} \in [0, 100]$. The results from fig.4.4 and 4.5 give us the optimal slack ratios shown in table 4.3.
We can observe that not only does the optimal slack ratio vary wildly between instances, but the cost of the initial solution can vary wildly with respect to $\frac{\alpha}{\beta}$. However we can see a global pattern: low values of $\frac{\alpha}{\beta}$ tend to be worse than high values, and it seems that the progression is asymptotic. All the variations seem

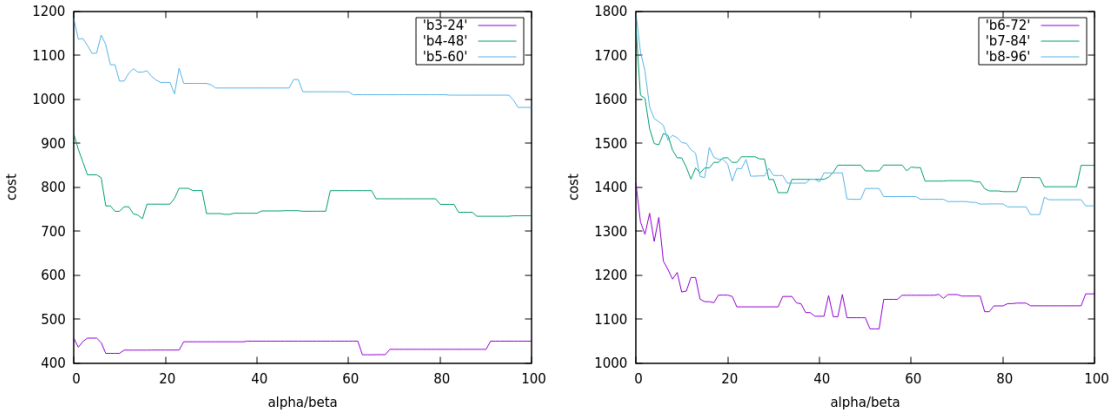

Figure 4.5: Cost of initial solution against $\frac{\alpha}{\beta}$, class b

instance	$\frac{\alpha}{\beta}$	instance	$\frac{\alpha}{\beta}$
a3-24	35	b3-24	63
a4-48	18	b4-48	15
a5-60	20	b5-60	97
a6-72	100	b6-72	51
a7-84	31	b7-84	31
a8-96	67	b8-96	86

Table 4.3: optimal slack ratios

to be due to the fact that the solution space is not continuous, thus there are limitations to which solutions are available for a given instance.

Let's try to confirm this asymptotic pattern by trying larger values, $\frac{\alpha}{\beta} \in [0, 10000]$. The results are shown in fig.4.6. These numbers do not confirm that larger slack ratios are better in general. In fact, for most of the instances the optimal ratios lie between 0 and 1000. Higher than that, we cannot get the best starting point. There are exceptions however. A minority of instances (ex. a4-48) tend to benefit from very large slack ratios and even get the best heuristic with $\beta = 0$.

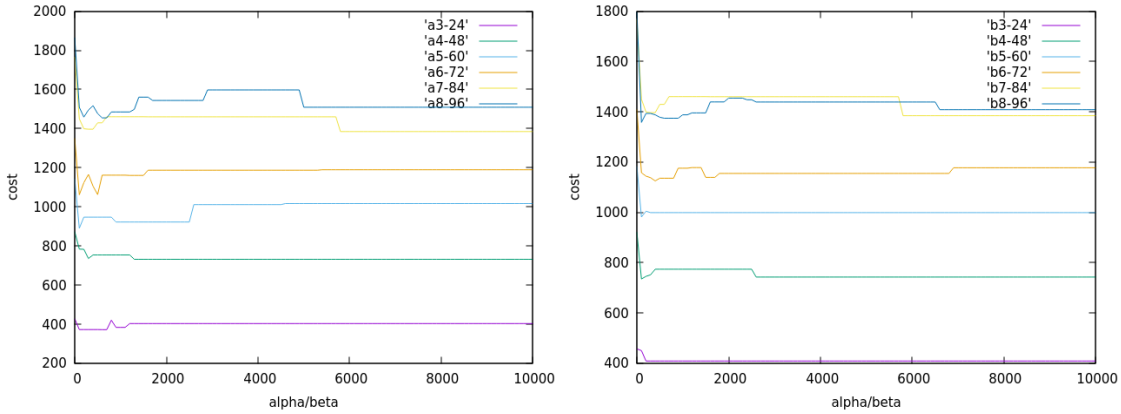

Figure 4.6: Cost of initial solution against $\frac{\alpha}{\beta}$, large values

What we can conclude from these results is that the choice of the slack ratio is very instance dependant, so if you want your solver to be as efficient as possible for a given DARP instance, then it might be a good idea to run a test similar to this one and find the optimal slack ratio for that instance before solving. But if you want a generalist approach that works well enough for many different instances, then using an arbitrary high slack ratio between 0 and 1000 or even setting β to 0 (thus not using the slack metric at all) are reasonable choices.

However, remember that this conclusion relies on the assumption that a good slack ratio for the initial solution is also a good slack ratio for the whole search. This might not be exact.

4.4 Experiment 4: Do ISVs influence performance?

In this test we measure and compare the performances of the 2 solvers to determine if the use of ISVs in the implementation causes any changes in efficiency. The goal here is not to beat state-of-the-art methods at finding good solutions, but to examine how the solver performs with or without ISVs on the same collection of DARP instances.

4.4.1 setup

In this test, every instance is solved by both solvers 15 times with a run time of 20 seconds, and the meta-parameters for the cost metric are set to $\alpha = 1$ and $\beta = 0$. The performance metric is the mean cost of the best solution found on each run. 20 seconds of run time may not seem like much, but as experiment 2 showed this is a problem with exponential complexity, thus most of the improving solutions are found during the first couple of seconds.

4.4.2 results & discussion

Class a		noISV		ISV	
m	n	mean cost	avg. fails	mean cost	avg. fails
3	24	353.86	28.53	353.53	24.8
3	30	510.88	292.8	515.02	219.27
4	32	514.07	200.2	518.13	139.93
4	40	605.92	137.13	608.68	118.0
4	48	705.00	85.4	704.83	71.4
5	50	752.71	191.53	757.12	123.0
6	60	918.40	134.93	924.60	114.4
7	84	1363.66	23.2	1377.10	17.01
8	96	1441.06	171.87	1480.7	100.0

Table 4.4: ISV vs noISV, Class a

The experiment indicates that the quality of the solutions is very close, but slightly worse on average for solver B.

The ISV approach performed about the same, if not slightly worse than the classic approach. But does this mean that ISVs are useless in this problem? By all means, no. As mentioned in previous chapters, the way both solvers handle some of the important constraints is essentially rendered equivalent by of one fact: when using this heuristic, the insertion points are grouped **by request**. Because

Class b		noISV		ISV	
m	n	mean cost	avg. fails	mean cost	avg. fails
3	24	401.36	0.0	400.78	0.0
3	30	546.92	5.13	550.19	3.33
4	32	512.66	7.87	514.16	5.87
4	40	675.90	7.73	675.65	7.8
4	48	714.03	15.2	719.00	7.73
5	50	816.71	2.13	813.39	1.67
6	60	952.14	18.0	949.47	11.73
7	84	1352.23	23.2	1386.66	18.67
8	96	1348.92	16.4	1374.13	7.47

Table 4.5: ISV vs noISV, Class b

of this, the filtering of the insertion points is the same whether it is done implicitly by ISVs and their constraints like in solver B, or explicitly like in solver A.
By consequence, the branching decisions are the same and we find the same solutions with both solvers. The filtering of the constraints is also a bit more computation-heavy when using ISVs, which explains why the ISV approach is a bit slower in the experimental results.
In conclusion, we can say that the thing holding back the ISVs in this experiment is not the LNS-FFPA search algorithm or MiniCP or the DARP itself, but the branching sheme. It is also worth mentioning that in terms of readability and interpretation, the usage of ISVs make the solver implementation a bit simpler and more intuitive.

4.5 Efficiency

This section is more a observation than an actual experiment. If we compare the results of experiment 4 with the competitive results of LNS-FFPA presented in [1], we can see quite a big difference in the performance of the solvers. This difference can be explained by several facts:

- The instances are not exactly the same. For instance in [8], the capacity of a vehicle is equal to 6 and the maximum ride time L is equal to 90 in all instances. This is not the case in the instances used in this paper, with often smaller values for these parameters.

- As mentioned in section 3.2.2, the search for insertion points has been modified to look for the pickup vertex then the delivery vertex, instead of looking for

the critical then non-critical vertex. This can result in a longer computation of the insertion points and an overall decrease in performance.

- The search time is much shorter in experiment 4 than in the experiments in [1].
- The meta-parameters of the search α, β don't have the same values.

Thus the results of [1] are not comparable with the results of this paper, which does not aim to match the original performances of LNS-FFPA anyways. This is just a quick disclaimer.

Chapter 5

Conclusion & improvement ideas

The final conclusion of this paper is that insertion sequence variables do not improve the performances of a DARP solver that implements the exact LNS-FFPA algorithm conceived by P. Van Hentenryck and S. Jain in [1]. To be more accurate, they fill the exact same role as successor arrays do, but in a slightly heavier and slower way. However, one positive aspect of insertion sequence variables is that they make the code of the solver more compact and readable because their high-level structure is a better fit for constraint programming in general.
Therewith, this paper only scratches the surface of the subject of using the ISV as a programming tool for solving the DARP. There are many limitations in the solvers presented, which are as many candidates for degrees of improvement in performance. Here are a few ideas for addressing some of these problems.

5.1 Search heuristic

As mentioned in the discussion on experiment 4, the search heuristic and the overall branching scheme impose a limit on what can be done with ISVs in solver B.
If the branching scheme is the problem, why not change it? If we defined insertion points by stop instead of by request, an insertion point pt for a vertex v_j would take the form $< v, p, e >$ where v is the vehicle visiting the stop, v_p its predecessor in the route and e the modified insertion cost, defined as

$$e(j, pt) = \alpha * costIncrease - \beta * gap_j \tag{5.1}$$

Where $costIncrease$ is the added route cost due to the insertion of v_j after v_p in **route**[**v**] and gap_i is defined by equation 2.5.

When using ISVs these new insertion points would be much simpler to compute, since we would just have to explore the insertion sets I of each ISV. The real

filtering work would be done by the constraints *Dependency*, *Precedence*, *TransitionTimes* and *Cumulative* as described in section 1.4., and these constraints we would probably need to be fully used and implemented.
With this change in mind, one could conceive modified versions of *TreeSearch* and *GetUnassignedRequest* designed to select and insert single stops instead of entire requests at the time. The actual LNS-FFPA implementation in *MinimizeRoutingCost* would not change at all.
As an additional perk, we would not need to worry about which vertex of a request to search first (critical or non-critcal? Pickup or delivery?) when computing the insertion points anymore. It would just be a matter of retrieving and sorting all the insertions of the routes according to the same criteria used by the current *GetUnassignedRequest*. This could make the heuristic more resilient and effective, and its implementation in the solver much easier.

5.2 Search parameters

As was demonstrated by some of the experiments above, choosing the right values for the parameters α, β, $range$, d and $numIter$ of LNS-FFPA is a tricky task with no perfect answer, and it is very instance-sensitive. It would certainly be very nice to have a program that takes a DARP instance as input and returns some values for the parameters as output. Maybe a trained machine learning model or a well chosen set of rules could perform that task.

5.3 Insertion propagation vs filtered search

In both of the solvers described in this paper, some of the DARP constraints are handled by propagation as a reaction to the insertion of vertices in the routes (method 1), while other constraints are handled by filtering out bad candidates for insertion points as they are found (method 2).

But the method used for filtering out a bad insertion point p has an influence on the heuristics of *TreeSearch* and *GetUnassignedRequest*. With method 1, p will be stored in **insertionObjChange** and count towards the heuristics, before being inserted and causing a failure. With method 2, p will not be considered at all as a potential insertion point and will not count towards the heuristics.

This effect may not be negligible because the presence of p in **insertionObjChange** can influence which request *TreeSearch* will branch on.
It is likely that this effect is the main reason behind the differences in branching

choices between solvers A and B, because both don't handle all of the constraints using the same methods.
It might be a good idea to study the impact of the effect theoretically or experimentally.

Bibliography

[1] S. Jain and P. Van Hentenryck, "Large neighborhood search for dial-a-ride problems," pp. 400–413, 01 2011. [Online]. Available: https://doi.org/10.1007/978-3-642-23786-7_31

[2] L. Michel, P. Schaus, and P. Van Hentenryck, "Minicp: a lightweight solver for constraint programming," *Mathematical Programming Computation*, vol. 13, no. 1, pp. 133–184, 2021. [Online]. Available: https://doi.org/10.1007/s12532-020-00190-7

[3] C. Thomas, R. Kameugne, and P. Schaus, "Insertion sequence variables for hybrid routing and scheduling problems," pp. 457–474, 09 2020. [Online]. Available: https://doi.org/10.1007/978-3-030-58942-4_30

[4] [Online]. Available: http://minicp.org

[5] [Online]. Available: https://minicp.bitbucket.io/apidocs/

[6] [Online]. Available: https://github.com/QDelmelle/minicp/tree/main

[7] [Online]. Available: https://bitbucket.org/oscarlib/oscar/wiki/CP

[8] J.-F. Cordeau and G. Laporte, "A tabu search heuristic for the static multi-vehicle dial-a-ride problem," 2002. [Online]. Available: https://doi.org/10.1016/S0191-2615(02)00045-0

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**