Item Recovery Problem
An ALNS-based Approach

João Resende[1], José Maravalhas-Silva[2]
Doctoral Program in Electrical and Computer Engineering,
Faculty of Engineering, University of Porto,
Porto, Portugal
Email: up202003528@fe.up.pt[1], up202003505@fe.up.pt[2]

***Abstract*—This paper presents the implementation of the Adaptive Large Neighboorhood Search (ALNS) heuristic in solving the item recovery problem. This problem consists in optimizing the path taken by an Autonomous Underwater Vehicle (AUV) whose goal is to retrieve items from several different underwater sites.**

***Index Terms*—Heuristics; Optimization; ALNS; AUV;**

I. Introduction

In the context of the course unit "Heuristics and Metaheuristics" from the "Doctoral Program in Electrical and Computer Engineering" from the Faculty of Engineering, University of Porto (FEUP), this paper presents the implementation of an heuristic, known as the Adaptive Large Neighboorhood Search (ALNS) [1], [2], and its usage in solving an optimization problem inspired by currently ongoing projects at the Centre for Robotics and Autonomous Systems (CRAS) belonging to the Institute for Systems and Computer Engineering, Technology and Science (INESCTEC).

The problem comprises the optimization of the path taken by an AUV in such a way that the AUV is able to pick up and retrieve different items which are spread throughout multiple sites across the seabead. To the best of our knowledge, there is no literature that features this exact problem, thus we will name it the "item recovery problem".

II. Problem Statement

The following list defines the details and constrains of the item recovery problem:

- The AUV starts at a "base" site
- There are multiple underwater sites that contain zero or more items that need to be "recovered" by the AUV
- Every item has a given weight
- The base site contains no items to be recovered
- The AUV has a maximum carrying capacity
- No item can exceed the maximum carrying capacity of the AUV by itself
- When the AUV is at the base site, all items that it is currently carrying are unloaded (i.e. they are "recovered")
- The AUV cannot unload items, except at the base site
- A weighted graph is used to represent the sites and the distances between them
- The goal is to recover all items while minimizing the total distance traveled

Figure 1 illustrates a possible scenario of this problem, where an AUV is used to recover items from an underwater cave.

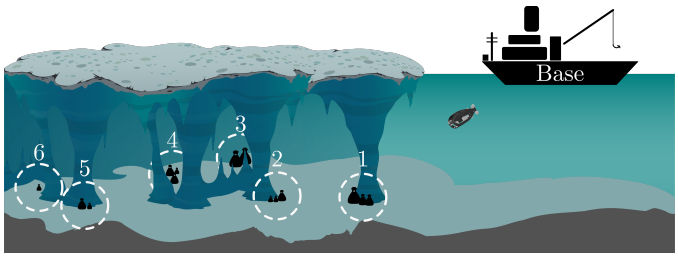

Fig. 1. Example of the item recovery problem.

The corresponding weighted graph is shown in Figure 2.

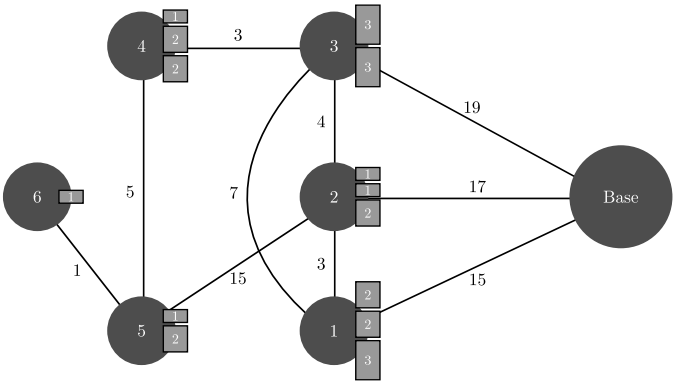

Fig. 2. Weighted graph of the scenario shown in Figure 1. The items and their weights are shown to the right of each site.

In Figure 2, the items and their weights are shown to the right of each site. For instance, site number 6 contains 2 items, one with a weight of 1 and another item with a weight of 2.

III. Solution Representation

The solution representation we used for this problem consists of a list of sites that the AUV visits. For each site, there is also a corresponding list of which items are picked.

A partial example of a solution for the scenario presented in Figures 1 and 2 is shown in Table I. This example assumes that items are indexed from top to bottom in Figure 2, and that the maximum carry weight is 4.

TABLE I
EXAMPLE OF A SOLUTION

Path	Base	1	2	1	Base	3	4	3	Base	...
Items Picked	[ ]	[0]	[0]	[ ]	[ ]	[ ]	[0,1]	[ ]	[ ]	...

As per the constrains of the problem, the AUV starts at the base site, and then moves to site 1 and picks up item 0, which has a weight of 2, as shown in Figure 2. It then moves to site 2 to pick up item 2, which also has a weight of 2. Having a full cargo, the AUV can no longer pick up any more items, and goes through site 1 to reach the base site and complete the retrieval. Looking at the graph and its weights, one obvious optimization to apply to this part of the solution would be to go directly to the base site from site 2.

IV. Optimization Technique

The proposed optimization technique is based on the Adaptive Large Neighborhood Search (ALNS) heuristic. In this section, we present both the generic implementation of the ALNS heuristic as well as our own procedures that are then fed to the ALNS-based optimization algorithm.

A. ALNS Heuristic

The first appearance of the ALNS heuristic was proposed in [2] in order to solve a pickup and delivery problem with time windows. This heuristic is an extension of the Large Neighborhood Search (LNS) heuristic [3]. Whereas the original LNS heuristic repeatedly applies a single destruction method and a single repair method, the ALNS heuristic uses pools of destruction and repair methods and dynamically updates the methods used in each iteration according to their past successes and failures.

The means by which destruction and repair methods are selected is via weights that define their probability of being chosen for the current iteration of the algorithm. The weights are then dynamically adjusted at the end of the iteration [1].

Algorithm 1 Adaptive Large Neighborhood Search [1].

1: **input** a feasible solution x
2: $x^b = x$; $\rho^- = (1, ..., 1)$; $\rho^+ = (1, ..., 1)$
3: **repeat**
4: Select a destruction method $d \in \Omega^-$ according to ρ^-
5: Select a repair method $r \in \Omega^+$ according to ρ^+
6: $x^t = r(d(x))$
7: **if** accept(x^t,x) **then**
8: $x = x^t$
9: **if** $cost(x^t) < cost(x^b)$ **then**
10: $x^b = x^t$
11: Update ρ^- and ρ^+;
12: **until** stop criterion is met
13: **return** x^b

Algorithm 1 demonstrates the high-level implementation of the ALNS heuristic. In this algorithm, x^t is the solution of the current iteration, x^b is the best known solution, and Ω^- and Ω^+ are the pools (i.e. the sets) of destroy and repair methods, respectively. Destruction and repair methods have associated weights ρ^- (for destruction methods) and ρ^+ (for repair methods), such that $\rho^- \in \mathbb{R}^{|\Omega^-|}$ and $\rho^+ \in \mathbb{R}^{|\Omega^+|}$.

In order to choose the methods in lines 4 and 5, a roulette wheel selection method is used. The probability $\varphi_j^\pm$ of selecting any given destruction (φ_j^-) or repair (φ_j^+) method j belonging to $\Omega^\pm$ is given by:

$$\varphi_j^\pm = \frac{\rho_j^\pm}{\sum_{i=1}^{|\Omega^\pm|} \rho_i^\pm} \tag{1}$$

For the dynamic update of the weights at the end of each iteration (line 13), a score Ψ is computed according to the following equation:

$$\Psi = max \begin{cases} \omega_1, \text{if the new solution is a new global best.} \\ \omega_2, \text{if the new solution is better than the current.} \\ \omega_3, \text{if the new solution is accepted.} \\ \omega_4, \text{if the new solution is rejected.} \end{cases}$$

where ω_1, ω_2, ω_3 and ω_4 are non-negative constants.

For both the destruction and repair methods used in the current iteration, their respective weights are updated as follows:

$$\rho_i^\pm = \lambda \rho_i^\pm + (1 - \lambda)\Psi \tag{2}$$

where $\lambda \in [0, 1]$ is the decay parameter that controls how much influence past scores have in relation to the current score Ψ. Notice how equation 2 effectively resembles the backward Euler discretization of a first order low-pass filter.

B. Destruction Methods

This subsection contains all the implementations and illustrative examples of the destruction methods we used to solve the item recovery problem.

1) Random Position Removal: This destruction method removes anywhere between 10% and 60% of all positions along the solution's path.

Algorithm 2 demonstrates exactly how this is implemented. Note how the algorithm includes and additional check to ensure that the solution is never empty.

An example of random positions being removed from a solution is shown in Figure 3.

Algorithm 2 Random Position Removal

1: **input** solution x
2: N = random(0.1, 0.6) * size(x)
3: N = max(1, floor(N))
4: **while** N != 0 and size(x) > 1 **do**
5: Remove a random position from x
6: N = N - 1
7: **return** solution x

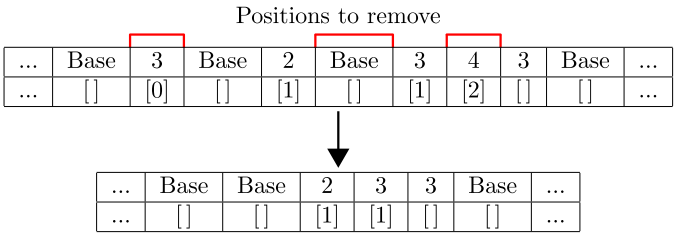

Fig. 3. Example of random positions being removed from a solution.

2) Random Position Swap: This destruction method randomly swaps (at most) 20% of all positions along the solution's path. Picked up items are also swapped along with their respective positions.

Algorithm 3 demonstrates the implementation of this destruction method, and an example is shown in Figure 4.

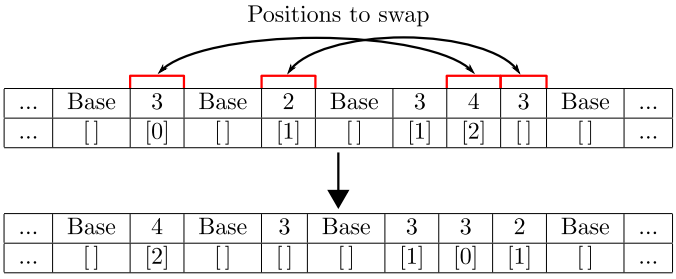

Fig. 4. Example of random positions being swapped in a solution.

Algorithm 3 Random Positions Swap

```
input solution x
N = 0.2 * size(x)
N = max(1, floor(N))
while N != 0 do
    Swap two random positions in solution x
    N = N - 1
return solution x
```

3) Random Subpath Removal: This destruction method randomly removes between 10% and 60% of all subpaths. A subpath is any part of a solution contained between two visits to the base site. It should be noted that when removing a subpath, a visit to the base site must be left in its place in order not to leave other subpaths incomplete.

Algorithm 4 demonstrates the implementation of this destruction method, and an example is shown in Figure 5.

Algorithm 4 Random Subpaths Removal

```
input solution x
N = random(0.1, 0.6) * number_subpaths(x)
N = max(1, floor(N))
while N != 0 and number_subpaths(x) > 1 do
    Remove a random subpath from solution x
    N = N - 1
return solution x
```

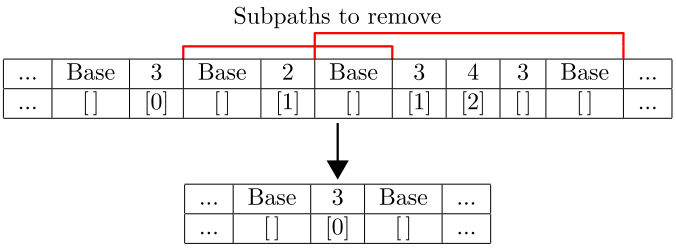

Fig. 5. Example of random subpaths being removed from a solution

4) Remove Worst Subpaths: In this destruction operator, all subpaths are ranked according to a "score" value. Each subpath's score S_i is based on how much item weight is retrieved during the subpath and how much cost does the subpath add to the overall solution:

$$S_i = \frac{\text{total item weight retrieved in subpath } i}{\text{cost of subpath } i} \tag{3}$$

After ranking all subpaths, between 10% and 60% of the worst scoring subpaths are removed. Algorithm 5 demonstrates the implementation of this destruction method, and an example is shown in Figure 6. The costs and item weights are taken from the scenario shown in Figure 2.

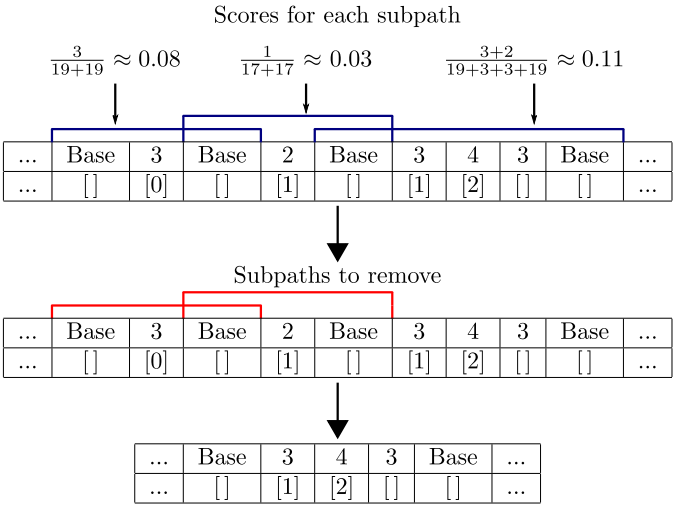

Fig. 6. Example of the worst subpaths being removed from a solution

Algorithm 5 Worst Subpaths Removal

```
input solution x
for each subpath i in solution x do
    Compute S_i
N = random(0.1, 0.6) * number_subpaths(x)
N = max(1, floor(N))
while N != 0 and number_subpaths(x) > 1 do
    Remove subpath i with lowest S_i from solution x
    N = N - 1
return solution x
```

C. Repair Method

Since repairing a solution for the item recovery problem is a particularly involved process, only a single repair method has been developed. This method is divided into multiple steps that progressively repair different aspects of the solution.

1) First Step: A valid solution must always begin at the base, and end at the base. To ensure this, the first step checks if the solution starts at the base, and merely inserts it at the begin of the solution's path if it is not present.

Similarly, if the solution does not end at the base, the solution is truncated - i.e. the last position is removed from the solution until the last position of the path is the base.

Algorithm 6 demonstrates how this repair step is implemented.

Algorithm 6 First Step of the Repair Method

```
1: input solution x
2: if Solution does not start at the base then
3:     Insert "Base" at the beginning of solution x
4: while Solution does not end at the base do
5:     Remove last position from solution x
```

2) Second Step: A valid solution must never have "invalid" moves (i.e. edges on the weighted graph that do not exist) along the path, nor should it have consecutive visits to the same site as these do not add any useful information.

The second step of the repair method traverses the path of the solution and inserts the shortest path between two sites whenever direct movement between said sites is impossible. The shortest path is computed using Dijkstra's algorithm.

Additionally, this step also merges any consecutive visits to the same site that may occur along the solution, along with their lists of picked up items.

Algorithm 7 demonstrates how this step is implemented.

Algorithm 7 Second Step of the Repair Method

```
1: input solution x
2: i = 0
3: while i < size(x)−1 do
4:     if site at x[i] == site at x[i + 1] then
5:         Add picked items from x[i + 1] to x[i]
6:         Delete x[i + 1]
7:     else if no connection between x[i] and x[i + 1] then
8:         repair_path = Dijkstra(Site at x[i], Site at x[i+1])
9:         Insert repair_path between x[i] and x[i + 1]
10:    else
11:        i = i + 1
```

It should be noted that in Algorithm 7, no items are picked up along $repair\_path$.

3) Third Step: A valid solution must never exceed the maximum carry weight. Thus, the third step of the repair method checks all subpaths within the solution, and, if necessary, randomly removes picked up items along each subpath that exceeds the carry weight until it is no longer exceeded.

Algorithm 8 demonstrates how this repair step is implemented.

Algorithm 8 Third Step of the Repair Method

```
1: input solution x
2: for each subpath in solution x do
3:     while max_carry_weight in subpath is exceeded do
4:         Remove a random item from a random position
           in subpath
```

4) Fourth Step: After the third step, the solution will likely be missing multiple items that still need to be recovered.

The fourth step of the repair procedure attempts to perform zero-cost insertions of the missing items in the current solution.

Algorithm 9 demonstrates how the zero-cost insertions are performed.

Algorithm 9 Fourth Step of the Repair Method

```
1: input solution x
2: for each item i at site j that has not been recovered do
3:     candidates = all indexes of x where site j is visited
4:     while item i at site j has not been recovered
5:           and size(candidates) != 0 do
6:         candidate = select random value from candidates
7:         subpath = subpath where candidate is contained
8:         if total item weight recovered in subpath + weight
9:           of item i at site j <= max_carry_weight then
10:            Insert item i (from site j) in x[candidate]
11:        else
12:            Remove candidate from candidates
```

5) Fifth Step: After the previous step, any items that have not yet been recovered can not be inserted at zero-cost in the current solution. Therefore, this step now appends new subpaths to the solution to pick up the remaining items.

Algorithm 10 demonstrates how this repair step is implemented.

Algorithm 10 Fifth Step of the Repair Method

```
1: input solution x
2: while not all items have been recovered do
3:     item = random item that has not been recovered
4:     site = site where item is located
5:     subpath = Dijkstra(Base, site) + Dijkstra(site, Base)
6:     Append subpath to solution x, ensuring that item
       is picked up along the way
```

It should be noted that in Algorithm 10, line 5's implementation should not result in consecutive visits to the same site.

6) Sixth Step: Technically, the sixth step is not necessary as the solution is guaranteed to be valid after the previous step. However, since the solution might contain subpaths where items are not picked up, removing them can help the intensification of the search algorithm, at the expense of

diversification. This is not a problem, since the ALNS heuristic is already extremely diversified, as the destruction methods remove significant portions of the solution.

Algorithm 11 demonstrates how the final post-processing step is implemented.

Algorithm 11 Sixth Step of the Repair Method

```
input solution x
for each subpath in x do
    if no items are picked along subpath then
        Remove subpath from solution x
```

D. Initial Solution

To generate the initial solution necessary for the ALNS heuristic, the repair method is called for a solution that has a single position, which is the base. This will make it such that all steps from 1 to 4 will fail to do anything, and step 5 will then generate a subpath for each item to be recovered.

E. Acceptance Criterion

For the acceptance criterion, a simulated annealing technique with linear temperature cooldown was used.

V. Results

A. Dataset

Since there are no pre-existing datasets of instances of the item recovery problem, we've created our own dataset. Some general information about the dataset is shown in Table II.

This dataset contains instances with anywhere from 40 to 100 sites.

In each instance, the carry weight C is randomly selected in the range $[2, 10]$.

The maximum number of items per site (M_i) was randomized between $[5, 10]$ for each instance. Then, when generating said instance, the actual number of items in each site is randomly selected between $[0, M_i]$.

All item weights are also randomly selected. In each instance, these weights have a minimum of 1, and a randomly chosen maximum in the range $[2, C]$.

For the connections between sites, all sites are connected to at least 1 other site. The actual number of connection ranges between $[1, M_c]$, where M_c is randomly selected between $[2, 5]$ for each instance.

It should be noted that the dataset used does not necessarily represent real-world scenarios. In our dataset's instances, while the connections between sites are relatively sparse, they are still completely random. In real-world scenarios, these connections would, most likely, not be random, but rather possess some structure to them.

B. Optimization Results

Table III contains the optimization results for all instances of our dataset after 200 iterations of the search algorithm.

The initial temperature of the simulated annealing acceptance criterion was set to 100, with each iteration of the search algorithm reducing this value by 5, down to a minimum of 5.

TABLE II
General Information of the Dataset Used

Instance	Sites	Sites With Items	Cargo Size	Graph Edges
1	40	36	2	101
2	45	39	9	112
3	50	44	4	66
4	55	47	7	111
5	60	49	3	140
6	65	61	6	98
7	70	63	3	169
8	75	62	5	193
9	80	68	4	185
10	85	72	9	170
11	90	72	9	228
12	95	86	9	236
13	100	91	10	156

As an example, Figures 7 and 8 contain information on the best and accepted costs per iteration, as well as diagnostics for all methods, for instance 1. We've found that other instances also follows similar trends.

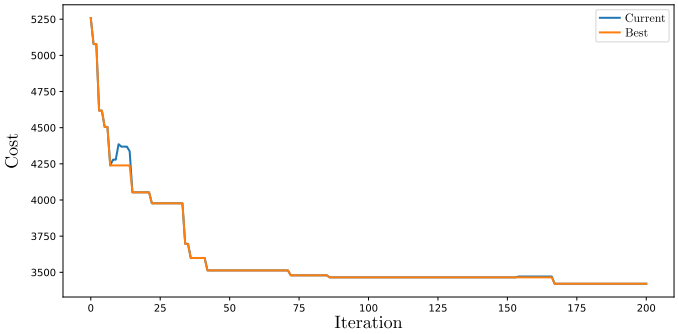

Fig. 7. Best and accepted Costs at each iteration - Instance 1

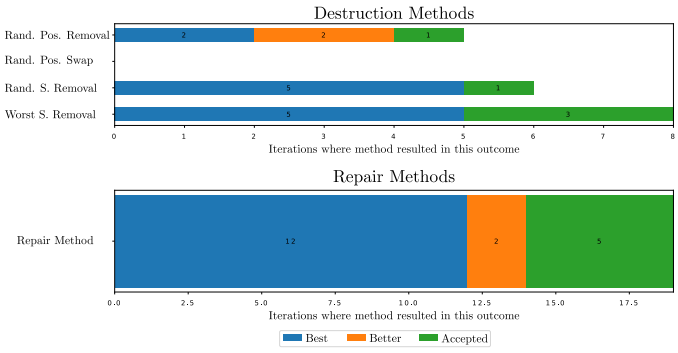

Fig. 8. Method diagnostics - Instance 1

TABLE III
OPTIMIZATION RESULTS

Instance	Time Taken	Initial Cost	Best Cost
1	28 min	5258	3421
2	1 h	5724	2856
3	1.3 h	9254	6046
4	2.1 h	13280	8072
5	1.6 h	8098	3797
6	9.7 h	16120	9928
7	6.7 h	6028	2818
8	5.5 h	11614	7354
9	5.5 h	8474	4040
10	9.4 h	7966	5018
11	2.2 h	7560	3125
12	10.5 h	20932	6192
13	51.6 h	35946	23444

VI. CONCLUSIONS

In this work we presented an optimization problem inspired by real underwater robotics projects and employed the ALNS heuristic to solve it.

The four destruction methods and the repair method implemented allowed for significant improvements over the initial solutions, but without knowing the optimal cost for each instance (or at least knowing some lower bound for it), it is difficult to ascertain whether or not our results are satisfactory.

Regarding the destruction methods used, it was clear from the method diagnostics from multiple instances that the ”random position swap” method was not very effective. In most instances, it never resulted in any improvement, only occasionally being accepted. If the acceptance criteria was a typical hill climbing criterion, then this method would have been useless every single time.

The ineffectiveness of the ”random position swap” is likely related to how it interacts with the repair method. Whenever a swap occurs, the solution will either remain valid, or it will have impossible connections between sites, depending on the sites being swapped and their neighbors. Consequently, the repair method may have to insert repair paths between the swapped sites and their neighbors, which will increase the overall's solution cost. If the swaps end up exceeding the carry weight on some supaths, some items will then need to be removed, which could lead to the repair method having to append additional subpaths to recover the remaining items that cannot be inserted back into the solution at zero-cost.

As for the time taken to process 200 steps of the search algorithm, we believe significant performance gains could be achieved by carefully optimizing our source code. While the number of iterations was kept to a small value to reduce the time necessary to run the algorithm, it's clear that many more iterations were needed to find better solutions, particularly on the instances with more sites.

Another possible future improvement to implement would be to dynamically update the degree of destruction - i.e. the percentage of the solution to be destroyed. This would likely entail the modification of the ALNS heuristic itself to include such a parameter, and research would need to be carried out to determine if this parameter should be specific to each destruction method, or shared between all destruction methods.

All source code and instances used can be accessed via our GitHub repository [4].

REFERENCES

[1] D. Pisinger and S. Ropke, *Large Neighborhood Search*. Springer US, 2010, pp. 399–419.

[2] S. Ropke and D. Pisinger, “An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows,” *Transportation Science*, 2006.

[3] P. Shaw, “Using constraint programming and local search methods to solve vehicle routing problems,” in *Principles and Practice of Constraint Programming — CP98*, M. Maher and J.-F. Puget, Eds. Springer Berlin Heidelberg, 1998.

[4] “Github repository.” [Online]. Available: https://github.com/rereee3/Item-Recovery-Problem-ALNS