A Very Large Neighbourhood Search approach to the Swath Segment Selection Problem

Roberto Cordone, Daniel Dissegna

Università degli Studi di Milano, Dipartimento di Informatica, Milano, Italy

roberto.cordone@unimi.it,daniel.dissegna@unimi.it

Keywords : *Swath Segment Selection, Knapsack problem, Very Large Neighbourhood Search.*

1 Introduction

The *Swath Segment Selection Problem* (*SSSP*) arises from the planning of Earth observations performed by satellites in quasi-polar orbits. As their movement combines with the Earth rotation, such satellites observe stripes of surface (*swaths*), that follow a north-east to south-west path during the descending semi-orbits and a south-east to north-west path during the ascending ones (see Figure 1). Each swath consists of a sequence of disjoint observation opportunities (*segments*), that can overlap with segments of other swaths. The portion of a target belonging to one or more overlapping segments is named *shard*. The *SSSP* amounts to deciding which segments to select, considering that the images collected in a swath are stored in a limited memory device and downlinked to ground stations at the end of each swath. The memory required by a shard depends on the segment used to collect it, due to the different extension of the shard along the main direction of the swath. Different shards yield different rewards. Collecting the same shard more than once is useless.

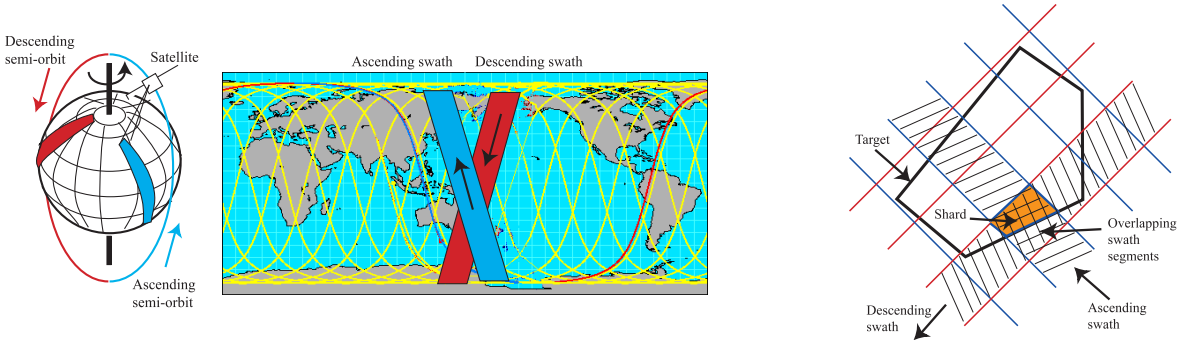

FIG. 1: Basic definitions: swaths, segments and shards

Muraoka [4] introduced the *SSSP* and proposed a constructive heuristic (*ASTER*) to solve it. Knight and Smith [3] developed a branch-and-bound algorithm, computing upper bounds with a network flow relaxation. Cordone et al. [1] implemented a branch-and-bound based on a Lagrangean relaxation of an Integer Linear Programming (*ILP*) formulation. Lagrangean relaxations and decompositions were discussed in [2]. Perea et al. [5] applied Constraint Programming to a variant where all shards must be collected at minimum cost.

Let S be the set of swaths, b_s the capacity of memory along swath $s \in S$, H the set of shards, r_h the reward gained collecting shard $h \in H$. The set of all segments N is partitioned into disjoint subsets $\bar{N}_s$ of segments belonging to each swath $s \in S$ or into disjoint subsets $\hat{N}_h$ of segments covering each shard $h \in H$. Conversely, h_i is the shard covered by segment i and i_{sh} the segment in swath s covering shard h (if there is any). $\hat{S}_h \subseteq S$ is the subset of swaths

covering shard $h \in H$. The memory required to select segment $i \in N$ is a_i. Formulation (1-5) uses binary variables x_i and y_h to indicate, respectively, whether segment $i \in N$ is selected or not and shard $h \in H$ is collected or not.

$$SSSP: \quad \max z = \sum_{h \in H} r_h y_h \tag{1}$$

$$\text{s.t.} \sum_{i \in \bar{N}_s} a_i x_i \leq b_s \qquad \forall s \in S \tag{2}$$

$$y_h \leq \sum_{i \in \hat{N}_h} x_i \qquad \forall h \in H \tag{3}$$

$$y_h \in \{0,1\} \qquad \forall h \in H \tag{4}$$

$$x_i \in \{0,1\} \qquad \forall i \in N \tag{5}$$

The objective (1) maximizes the reward given by all collected shards. Constraints (2) restrict the segments selected in each swath. Constraints (3) state that a shard is collected only selecting one of the segments that cover it. Since collecting a shard multiple times is useless, they can be turned into equalities.

2 The VLNS algorithm

We here describe a Very Large Neighbourhood Search (*VLNS*) approach, based on the search for sequences of variable modifications in Formulation (1-5). To build a starting solution, we assign to each segment i the reward of the shard h_i it covers according to (3), and solve the knapsack problems (2) related to the swaths in a given order, avoiding to select segments that cover shards collected by previous swaths. As swaths in the same direction overlap less (if ever), we apply two orders: first the ascending ones then the descending ones, and vice versa.

The *VLNS* approach (see Algorithm 1) receives this feasible starting solution (x, y) and returns an improved one (x^*, y^*). For the sake of simplicity, all following pseudocodes consider the instance data as directly accessible. Procedure LocalSearch looks for a feasible improving solution in an exponential-size neighborhood, composed by the solutions that collect a new shard, uncollect at most one, and swap the segments that cover $\lambda \geq 0$ currently collected shards. If the new solution improves the best known, parameter λ goes back to 1; otherwise, it increases to consider other solutions. The computational effort is limited restricting the number of segment swaps ($\lambda_{\max} \leq |H|$), the overall computational time ($T_{\max}$) and the number of uncollected shards considered ($\mu_{\max} \leq |H|$).

```
1  Algorithm VLNS(x, y, λmax, Tmax, μmax)
2  |  (x*, y*) := (x, y); λ := 1;
3  |  do
4  |  |  (x, y) := LocalSearch(x, y, λ, μmax);
5  |  |  if z(x, y) > z(x*, y*) then
6  |  |  |  λ := 1; (x*, y*) := (x, y);
7  |  |  else
8  |  |  |  λ := λ + 1;
9  |  |  end
10 |  while (λ <= λmax) or (time > Tmax);
11 |  return (x*, y*)
```

Algorithm 1: The *VLNS* algorithm

The LocalSearch procedure (see Algorithm 2) builds the set $\bar{H}$ of shards currently uncollected. As long as this set is nonempty and the number of shards considered does not exceed $\mu_{\max}$, it extracts from $\bar{H}$ the shard h^* with maximum reward. For every swath that covers it, the procedure marks all shards as modifiable and calls function CollectShard to find a solution (x', y') covering h^* with swath s. The first improving solution found is returned.

```
1  Function LocalSearch(x, y, λ, μ_max)
2      H̄ := {h ∈ H : y_h = 0}; μ := 0;
3      while (H̄ ≠ ∅) and (μ < μ_max) do
4          h* := arg max_{h∈H̄} r_h; H̄ := H̄ \ {h*}; μ := μ + 1
5          for s ∈ S̄_{h*} do
6              for h ∈ H do
7                  v_h := false;
8              end
9              (x', y') := CollectShard(x, y, r_{h*}, h*, s, v, λ);
10             if z(x', y') > z(x, y) then return (x', y'); ;
11         end
12     end
13     return (x, y);
```

Algorithm 2: The *LocalSearch* procedure

The CollectShard procedure (see Algorithm 3) looks for a chain of at most λ variable modifications that gains a reward r^* collecting shard h along swath s. The procedure is recursive. In the base case ($\lambda < 0$), the problem has no solution. Otherwise, we increase y_h and $x_{i_{sh}}$ to collect shard h using the only segment in swath s that covers it. If the resulting solution is feasible, we directly return it. Otherwise, we mark the shard as nonmodifiable, and scan all segments in swath s ($j \in \bar{N}_s$) that collect a modifiable shard ($x_j = 1 \wedge v_{h_j} = \text{false}$) whose removal yields a feasible solution. If the shard h_j has a reward smaller than r^*, we get a feasible and improved solution, and simply return it. Otherwise, we try to recollect h_j with a new swath s' by recursively calling CollectShard. The process generates a chain of variable modifications that increase a variable y, alternatively decrease and increase $\lambda \geq 0$ variables x and possibly decrease a variable y.

```
1  Function CollectShard(x, y, r*, h, s, v, λ)
2      if λ < 0 then return (x, y) ;
3      y_h := 1; x_{i_sh} := 1;
4      if Σ_{i∈N̄_s} a_i x_i ≤ b_s then return (x, y) ;
5      v_h := true;
6      for j ∈ N̂_s : x_j = 1 ∧ v_{h_j} = false ∧ Σ_{i∈N̄_s} a_i x_i − a_j ≤ b_s do
7          x_j := 0;
8          if r_{h_j} < r* then
9              y_{h_j} := 0;
10             return (x, y);
11         else
12             for s' ∈ S̄ \ {s} do
13                 return CollectShard(x, y, r*, h_j, s', v, λ − 1);
14             end
15         end
16         x_j := 1;
17     end
18     return (x, y);
```

Algorithm 3: The *CollectShard* procedure

3 Results

The benchmark instances of [1] consider n ascending and n descending swaths, with 14 values of n ranging from 10 to 500. Each shard is covered by a segment of each group of swaths

($|\hat{N}_h| = 2$) and its reward is drawn from $\{1, \ldots, 100\}$ or $\{51, \ldots, 100\}$ with a uniform random distribution. The same holds for the memory consumption of each segment, but the values for the two segments covering the same shard are either independent or identical. The capacity of each swath is a fraction of the smallest total memory consumption along the swath: $b_s = \alpha \min_{s \in S} \sum_{i \in \bar{N}_s} a_i$, with $\alpha \in \{0.2, 0.3, 0.4\}$. Overall, there are $14 \cdot 2 \cdot 2 \cdot 2 \cdot 3 = 336$ instances.

We compare the *VLNS* approach with the root node of the branch-and-bound of [1], which applies 1 000 times a subgradient update and a Lagrangean heuristic. The execution time on a processor Intel Xeon E5-2620 2.1 GHz and 16 GB of RAM ranges from 0.01 to 216 seconds (12 seconds on average). We saved the time for each instance and imposed it as a time limit $T_{\max}$ on the *VLNS* algorithm, but also set $\lambda_{\max} = 20$ and $\mu_{\max} = 500$ to avoid spending time to explore unpromising sequences. In about one third of the cases, these limits or the discovery of a local optimum anticipated the termination, reducing the average time to 10 seconds.

In Table 1, the first column reports the size n. The following three ones provide (for the Lagrangean approach) the percent gaps of the heuristic solution and of the upper bound with respect to the best known result, and the running time in seconds. The last two columns provide the heuristic gap and the running time for the *VLNS* approach. The values are averaged over the 24 instances of each size. The *VLNS* approach outperforms the Lagrangean one, decreasing the average gap from 2.10% to 1.24% and improving 58 of the 271 best known solutions not provably optimal. *VLNS* seems a viable way to obtain good heuristic solutions in short time.

	Lagrangean approach			VLNS	
n	GapLB	GapUB	CPU	GapLB	CPU
10	6.10%	0.08%	0.030	1.77%	0.006
20	3.32%	0.59%	0.285	1.00%	0.200
30	2.38%	0.53%	0.625	0.90%	0.560
40	1.96%	0.53%	1.054	1.00%	0.995
50	1.84%	0.50%	1.504	1.12%	1.507
60	1.71%	0.48%	2.063	1.21%	1.991
70	1.65%	0.42%	2.658	1.07%	2.268
80	1.64%	0.43%	3.393	1.14%	3.182
90	1.56%	0.42%	4.124	1.21%	3.635
100	1.48%	0.39%	4.721	1.26%	4.138
200	1.46%	0.34%	14.039	1.37%	11.849
300	1.44%	0.33%	26.974	1.41%	23.289
400	1.46%	0.33%	43.526	1.43%	36.576
500	1.43%	0.32%	63.136	1.41%	54.455

TAB. 1: Average gaps and computational times for the Lagrangean and the *VLNS* approach

References

[1] R. Cordone, F. Gandellini, and G. Righini. Solving the swath segment selection problem through Lagrangean relaxation. *Comput Oper Res*, 35(3):854–862, March 2008.

[2] R. Cordone, G. Righini, and A. Taverna. Upper and lower bounds for the swath segment selection problem. In *Proc. of CTW17*, pages 53–56, Cologne, Germany, June 6–8 2017.

[3] R. Knight and B. Smith. Optimal nadir observation scheduling. In *Proc. of IWPSS 2004*, Darmstadt, Germany, June 23–25 2004. ESA–ESOC.

[4] H. Muraoka, R. H. Cohen, T. Ohno, and N. Doi. ASTER observation scheduling algorithm. In *Proceedings of SpaceOps 1998*, Tokio, Japan, June 1–5 1998.

[5] F. Perea, R. Vazquez, and J. Galán-Vioque. Swath-acquisition planning in multiple-satellite missions: An exact and heuristic approach. *IEEE Trans Aerosp Electron Syst*, 1(3):1717–1725, July 2015.