Orienteering - The Unesco Challenge

Marco Caserta

1 Problem Description

The Unesco Challenge is aimed at finding a maximum prize route across a set of sites without exceeding a maximum travelling time. In addition, a few side constraints are to be considered (e.g., balance of the solution in terms of type of sites visited) and by the fact that the prize associated to a site depends on some of the other sites included in the route.

A mathematical model for the problem can be sketched as follows. Let us define the following sets:

- P: set of candidate places, or sites ($j = 1, \ldots, |P|$). We indicate with $P' = P \cup \{0\}$ the set which includes home, labelled as site 0;
- S: set of countries ($k = 1, \ldots, |S|$);
- S_k : set of sites in country k;
- D : set of endangered sites;
- C, N, and M: set of cultural, natural, and mixed places.

In addition, we define the following variables:

- x_{ij} takes value 1 if site j is visited after site i and 0 otherwise, for $i, j \in P'$;
- y_j takes value 1 if site j is visited, and 0 otherwise, for $j \in P$;
- z_k takes value 1 if country k is visited and 0 otherwise, with $k \in S$.

Therefore, the problem can be formulated as:

$$(OP): \max \sum_{j \in P} y_j + \sum_{k \in S} 2 \times z_k + \sum_{j \in D} 3 \times y_i \tag{1}$$

$$\text{s.t.} \quad \sum_{i \in P'} \sum_{j \in P'} t_{ij} x_{ij} \leq B \tag{2}$$

$$\sum_{i \in P} x_{ij} \geq y_j, \quad j \in P \tag{3}$$

$$\sum_{j \in S_k} y_j \geq z_k, \quad k \in S \tag{4}$$

$$|\sum_{j \in C} y_j - \sum_{j \in N} y_j| \leq \sum_{j \in M} y_j + 1 \tag{5}$$

$$\text{subtour elimination constraints} \tag{6}$$

$$x_{ij} \in \{0, 1\}, \quad i, j \in P' \tag{7}$$

$$y_j \in \{0, 1\}, \quad j \in P \tag{8}$$

$$z_k \in \{0, 1\}, \quad k \in S \tag{9}$$

where t_{ij} indicates the travel time between site i and site j, and B is the available travel time.

The objective function maximizes the overall score of a solution, given by the weighted sum of the number of sites, the number of countries, and the number of visited endangered sites. Constraint (2) ensures that the maximum traveling time is not exceeded; constraints (3) ensure that a site is accounted for only if it is a destination of another point; constaints (4) guarantee that the country variable takes up a value of 1 only if at least one site of that country has been selected; finally, constraint (5) captures the satisfaction of the following property: an equal number of sites of type C and type N are visited, where sites of type M can be classified as needed. More precisely, in order not to restrict the domain of the solutions to routes with an even number of sites, the constraint enforces that the maximum difference between the number of sites of type C and N (after adjusting the classification of sites of type M) is at most 1. Later on, we will illustrate how this can easily be changed if an even number of sites must be ensured. The last three constraints define the nature of the variables, i.e., this model is a 0-1 (binary) program, since all the variables are binary. Finally, in order to ensure a valid tour (at least, in the TSP sense), subtour elimination constraints should be added here (e.g., the Miller-Tucker-Zemlin subtour elimination constraints.)

Two hypotheses, not explicitly mentioned in the assignment, have been introduced here:

- The first one is that constraint (5) is correct. The constraint presented above is a more general version of the requirement, in the sense that it allows a maximum inbalance of one

site between sites of type C and N. That is, it does not restrict the solution space to routes with an even number of sites. However, both the model and the associated code can easily be modified by removing the "+1" from the constraint.

- Subtours are not allowed, in the spirit of the Traveling Salesman Problem.

2 Solution Strategy and Algorithm

The model presented in the previous section resembles a well-known problem from the literature, the Orienteering problem, in which one has to select nodes from a graph to maximize the prize collected at each node without exceeding a budget (typically, a distance.) The problem studied here has the added complication provided by the balance constraint (5). Since the Orienteering problem is known to be $\mathcal{NP}$-Hard, we assume that (OP) is also $\mathcal{NP}$-Hard. This claim would require a rigorous prove (by reduction) but it seems reasonable at this point.

In addition, given that the objective function coefficient of many of the binary variables is the same, the reduced costs in a linear relaxation would tend to be quite similar and, therefore, not too informatives in terms of dominance across variables. It is reasonable to expect that the linear relaxation of (OP) will produce quite loose bounds. In light of these considerations, we did not attempt to use a mathematical programming-based technique to solve the problem and we decided to resort to (meta)-heuristic schemes.

The algorithm implemented for (OP) is based on a simple implementation of a metaheuristic known as Iterative Local Search (ILS). The ILS has been successfully employed in the context of the Orienteering problem. The basic structure of the algorithm is provided below in Algorithm 1.

The algorithm `ILS()` is therefore based on four heuristics, briefly described below:

- `insertion_heuristic()`: Starting from a solution that only contains the home site, we add, in a greedy fashion, sites to the current solution, as long as a site can be added without violating the budget constraint (2). In this phase, the balance constraint (5) is ignored. Therefore, at the end of this phase, the solution need not be feasible with respect to the balance constraint. Details on the greedy score are presented in the documentation of the code accompanying this document (under `docs/HTML`.)

- `repair_heuristic()`: Transforms the incumbent solution into a balanced one, enforcing the satisfaction of constraint (5), in two different ways:

  - swapping: remove from the current solution one site belonging to the class in excess (C or N), and add from the available solution a site of the opposite class. The swap is

Algorithm 1 : `ILS()`

Require: problem OP, stopping criteria, coordinates of home site
Ensure: best feasible route $\mathbf{r} = (\mathbf{x}, \mathbf{y}, \mathbf{z})$

$k = 0$ (iterations counter)
relax constraint (5) and build incumbent solution $\mathbf{r}^I$ with `insertion_heuristic()`
while not stopping criteria **do**
  $k \leftarrow k + 1$ (iteration counter)
  `no_improv` $= 0$ (nr. iterations without improvements)
  $\mathbf{r}^I \leftarrow$ `repair_heuristic(`$\mathbf{r}^I$`)` to re-introduce constraint (5)
  `stop = false`
  while `!stop` **do**
    relax (5) and $\mathbf{r} \leftarrow$ `2-Opt(`$\mathbf{r}^I$`)` heuristic to repaired incumbent
    $\mathbf{r} \leftarrow$ `repair_heuristic(`$\mathbf{r}$`)` to re-introduce constraint (5)
    if $\mathbf{r}$ is an improving solution **then**
      $\mathbf{r}^I \leftarrow \mathbf{r}$
      `no_improv` $= 0$
    else
      increament `no_improv` by one
      if max number of iterations without improvements has been reached, `stop = true`
    end if
  end while
  update best route $\mathbf{r}^*$ if needed
  $\mathbf{r}^I \leftarrow$ `random_perturbation(`$\mathbf{r}^I$`)` heuristic (remove % of sites in $\mathbf{r}^I$)
end while

selected in such a way that the change in the objective function is maximized, without violating the time limit constaint;

- random removal: if a feasible swap is not found and yet the solution does not satisfy constaint (5), a site belonging to the largest class is randomly selected and removed from the current solution.

This two-step approach is iteratively applied until a feasible solution is reached.

- `2-Opt()`: A classical 2-opt move is applied, in which a site that is currently in the solution is swapped with one not in the solution, with the aim of maximizing the increase in objective function value. This heuristic operates on a relaxed version of (OP), in which the balance constraint is removed. Therefore, after the 2-Opt scheme, the heuristic scheme must be applied again.

The four-heuristics scheme benefits from multiple restarts, due to the random mechanisms of `random_perturbation()` and `repair_heuristic()`. Therefore, as long as a predefined number of iterations has not been reached, we take the current solution and randomly remove up to a $\gamma\%$ of sites from the current route. We thus obtain a new incumbent, and we restart the cycle.

The stopping criteria are defined by two conditions, and we stop whenever the first is reached: either (i) a maximum number of iterations without improvements has been reached or (ii) a total maximum number of iterations has been completed.

3 Data Structure and Implementation Issues

A full description of the project structure is provided under the `docs` folder of the project. The file `index.html` under the `HTML` folder provides an overview of the organization of the project. Details about the implementations are thereby presented.

With respect to the implementation of algorithm `ILS()`, we defined the following classes:

- `Orienteering`: The main class, implementing all the heuristics of the ILS cycle and containing the basic data structure. To import the instance data, we created a `DataFrame` as a list of `Records`. Each record contains all the information available on a site (id, name, coordinates, country, etc.)

- `Move`: This class implements the different moves carried out by the heuristics, namely:

  - the insertion move of the greedy insertion heuristic;
  - the removal-and-insertion move of the repair heuristic;

  - the swap move of the 2-Opt heuristic;
  - the random removal of sites of the random perturbation heuristic.

- `Solution`: This class contains the data of a solution, i.e., the `inSet` (the set of sites in the current solution), `outSet` (the set of sites not in the current solution), the objective function value of the current solution `Z`, the total travel time `time`, the set of visited countries `countrySet`, etc. In addition, this class defines the actual updates on to the solution, i.e., how a new node is added to the route, how the 2-Opt swap is carried out.

We also defined auxiliary classes to parse the command line, as described in the project documentation, under the folder `docs`.

Observations. Since the algorithm is based on the exploration of multiple neighbhorhoods, we placed special care on the solution evaluation functions. More specifically, every time a new solution in a neighborhood is reached, such routes must be evaluated in terms of (i) objective function value, and (ii) time constraint. Rather than recomputing the objective function value and the time associated to a solution, we keep track of the variation of these two scores generated by each move. Therefore, the full computation of the value of the objective function and of the travel time is no longer needed, if a proper accounting of the moves is carried out throughout the local search.

Figure 1 illustrates a solution obtained using as home point the offices of the company in London. The solution is composed of 49 sites, 15 of these endengered sites, and 38 different countries. The objective function value is 170 and the total travel time is 30228 minutes (ouf of an available total of 30240 minutes), which corresponds to a trip of 16784 km. It is worth remembering that the travel time is the sum of two terms, the actual travel time from site to site and the visiting time (6 hours) of each place visited. On the other hand, the travel distance, in km, corresponds to the sum of the distances from site to site. This explains the discrepancy between the two numbers.

4 UnitTests

We designed tests for the following two categories:

- Tests for the evaluation functions: We thoroughtly tested that the objective function value and the time limit are properly computed. The class `TestFunctionsCorrectness` implements this type of tests.

- Tests for the correctness of the final solution. We want to ensure that the final solution complies with all the constraints of the model (including the subtour constraints.) These tests, when successful, guarantee that the final solution is actually feasible. The class `TestSolCorrectness` implements this pool of tests.

In addition, beyond the use of unittests, the code uses standard `assert()` clauses, which can be activated using the flag `-ea` in compilation. These are aimed at ensuring that the update of the objective function and of the travel time is properly carried out after each move.

5 Parameters of the Instance

- The instance `whc-sites-2019.xl` has been transformed, using a spreadsheet, into a `csv` file. The relevant portion of this file is found under the folder `data`, file `unesco.csv`. This is the file that is imported and used to compute the distances and the characteristics of each site. Details about the Haversine transformation function are provided in the code and its documentation.

- Travel budget B. This is obtained assuming a travel duration of 3 weeks (as stated in the specifications) and continuous travelling (since it is mentioned that sleeping time can be assumed to occur during traveling.) Therefore, we have:

$$B = 21 \times 24 \times 60 = 3024 \text{ minutes}$$

- Visit time: Specified as 6 hours per site, i.e., 360 minutes per site.

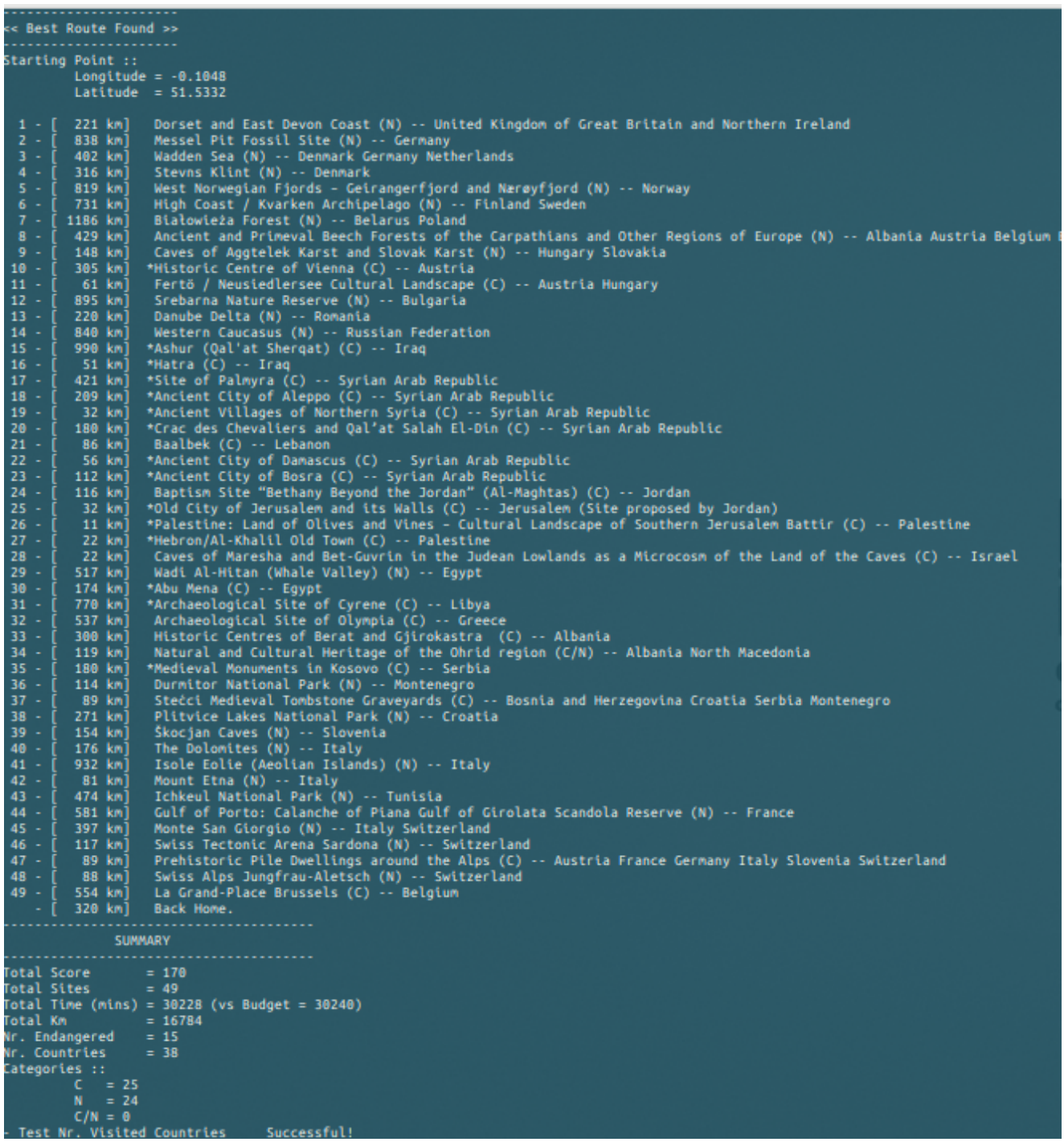

Figure 1: Example of a solution starting from London and visiting 49 sites, 38 countries.