Combinatorial Optimization
Lab No. 10
Traveling Salesman Problem

Industrial Informatics Research Center
http://industrialinformatics.cz/

May 29, 2018

Abstract

In this lab we review various ways how to solve the famous Traveling Salesman Problem. A new Integer Linear Programming formulation is introduced along with an approach based on lazy constraints. As a homework, you will solve a problem of image unshredding.

1 Traveling Salesman problem

Traveling Salesman Problem (TSP) is one of the most famous problems in combinatorial optimization. TSP has a wide range of applications in both theory and practice. For example, TSP is natural model for solving problems in logistics.

Let $G = (V, E)$ be a complete, directed graph where $V = \{1, \ldots, n\}$ is the set of nodes and $E = \{(i, j) \mid i, j \in V, i \neq j\}$ is the set of edges. We assume $n \geq 3$. Let each edge $(i, j) \in E$ be associated with cost $c_{i,j}$. The task is to find a closed path of minimal cost going through each node of G exactly once Hamiltonian circuit of a minimal cost). As an example, consider a graph from Fig. 1a; the corresponding optimal solution is depicted in Fig. 1b.
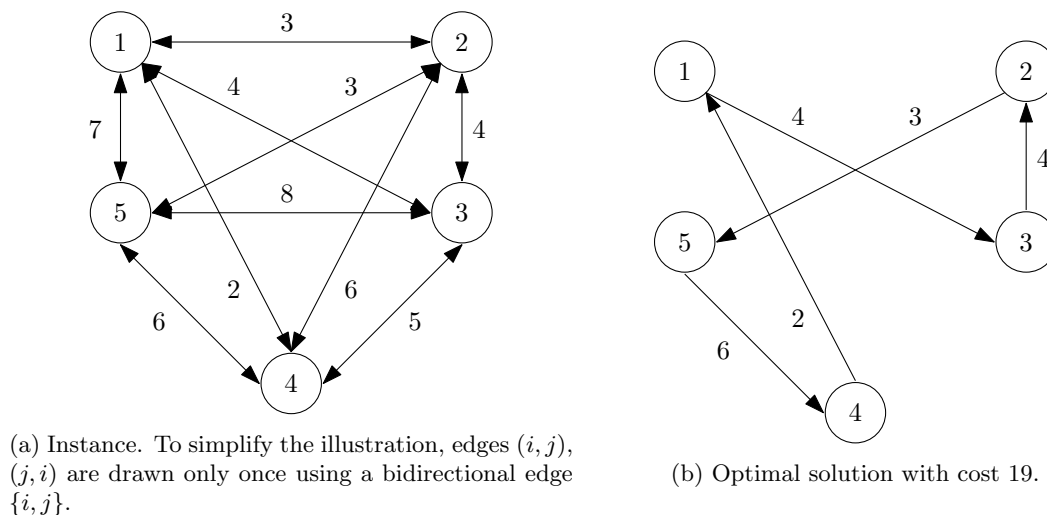

(a) Instance. To simplify the illustration, edges (i, j), (j, i) are drawn only once using a bidirectional edge $\{i, j\}$.

(b) Optimal solution with cost 19.

Figure 1: TSP example with symmetric costs.

On lectures, you were introduced to the following Integer Linear Programming (ILP) model of TSP

$$\min \quad \sum_{(i,j)\in E} c_{i,j} \cdot x_{i,j} \tag{1}$$

$$\text{s.t.} \quad \sum_{(i,j)\in E} x_{i,j} = 1, \quad j \in V \tag{2}$$

$$\sum_{(i,j)\in E} x_{i,j} = 1, \quad i \in V \tag{3}$$

$$s_i + c_{i,j} \leq s_j + M \cdot (1 - x_{i,j}), \quad i \in V, j \in V \setminus \{1\} \tag{4}$$

$$x_{i,j} \in \{0,1\}, \, s_{i,j} \in \mathbb{R} \tag{5}$$

Figure 2: ILP formulation of TSP from the lecture.

where $x_{i,j}$ is a binary variable such that $x_{i,j} = 1$ iff node i in the circuit just before node j. s_i is a variable representing the "time" of arrival into the node and is used for eliminating subtours.

Although this model is straightforward, we show a more efficient one below. Nevertheless, the model in Fig. 2 is still useful if we consider TSP with time-windows (time interval in which the node has to be visited).

1.1 More efficient ILP formulation for TSP

More efficient ILP model is shown in Fig. 3 where the decision variable $x_{i,j}$ indicates whether the edge $(i,j) \in E$ belongs to the circuit or not.

$$\min \quad \sum_{(i,j)\in E} c_{i,j} \cdot x_{i,j} \tag{6}$$

$$\text{s.t.} \quad \sum_{(i,j)\in E} x_{i,j} = 1, \quad j \in V \tag{7}$$

$$\sum_{(i,j)\in E} x_{i,j} = 1, \quad i \in V \tag{8}$$

$$\sum_{(i,j)\in E} y_{i,j} - 1 = \sum_{(j,k)\in E} y_{j,k}, \quad j \in V \setminus \{1\} \tag{9}$$

$$\sum_{(i,1)\in E} y_{i,1} + (n-1) = \sum_{(1,k)\in E} y_{1,k} \tag{10}$$

$$y_{i,j} \leq (n-1) \cdot x_{i,j}, \quad (i,j) \in E \tag{11}$$

$$x_{i,j} \in \{0,1\}, \, y_{i,j} \in \mathbb{Z}_{\geq 0} \tag{12}$$

Figure 3: More efficient ILP formulation of TSP.

One thing to mention about the model is the elimination of the subtours, see constraints (9)–(11). The idea is that we assume the salesman sells n items during his journey starting from node 1 and he is required to sell exactly one item in each node. Let $y_{i,j}$ be the number of items which the salesman has after leaving node i and before entering node j. Due to the selling requirement, the number of remaining items before visiting node $j \in V \setminus \{1\}$ is one more than the number of remaining items after leaving node j (see constraint (9)). Notice that when the salesman return back to its starting node 1, it must hold that $y_{i,1} = 0$, where i is the immediate predecessor to node 1 during the salesman tour. Thus constraint (9) is modified for this special case as shown in constraint (10). Constraint (11) denotes that the upper bound of $y_{i,j}$ is $n-1$ while

the lower bound is 0. It is obvious that when the edge $(i,j) \in E$ does not belong to the salesman tour, then $y_{i,j}$ equals 0. In Fig. 4a we show an example of a feasible solution whereas in Fig. 4b we show an example of an infeasible solution with multiple subtours that violates constraint (10).
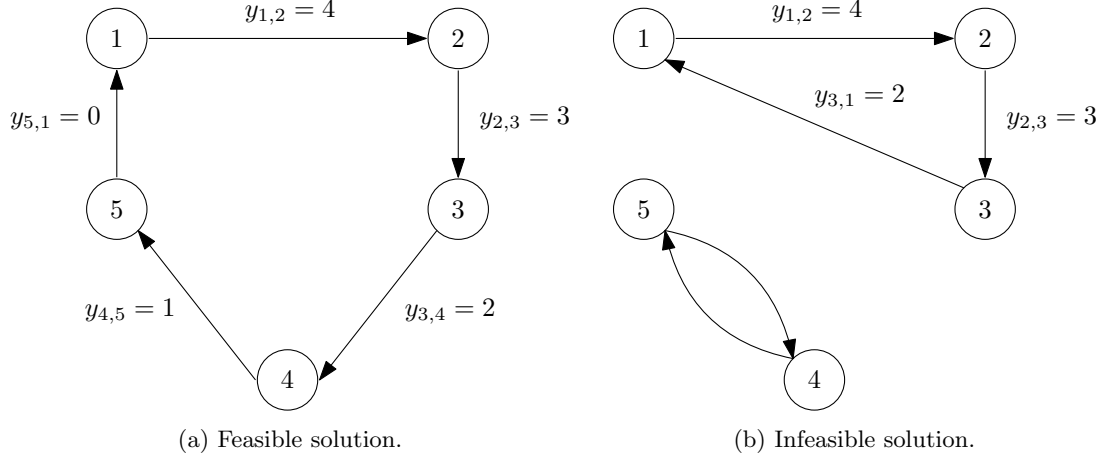

(a) Feasible solution.

(b) Infeasible solution.

Figure 4: Feasible and infeasible solutions to model in Fig. 3.

1.2 Lazy Constraints

Since combinatorial optimization problems are studied for decades already, it is natural that there exist a lot of advanced optimization approaches that result in reduced computation time for certain problem types. One of the conceptually easiest approach is called *lazy constraints* [1]. The basic idea is to initially formulate the problem only with the most essential constraints, omitting those that are only rarely violated. These other constraints are checked and added one-by-one to the model only if the current solution violates any of them. In other words, some constraints are generated in a lazy fashion, i.e. the constraint is added to the model only if the solution violates it.

The application of lazy constraints to TSP is really straightforward. Let model TSP as follows

$$\min \quad \sum_{(i,j)\in E} c_{i,j} \cdot x_{i,j} \tag{13}$$

$$\text{s.t.} \quad \sum_{(i,j)\in E} x_{i,j} = 1, \quad j \in V \tag{14}$$

$$\sum_{(i,j)\in E} x_{i,j} = 1, \quad i \in V \tag{15}$$

$$\sum_{i,j\in S: i\neq j} x_{i,j} \leq |S| - 1, \quad S \subset V, S \neq \emptyset \tag{16}$$

$$x_{i,j} \in \{0,1\} \tag{17}$$

Figure 5: ILP formulation of TSP, suitable for lazy constraints.

Notice that model in Fig. 5 is the same as in Fig. 3 with exception that constraints (9)-(11) were replaced with constraints (16) which eliminate subtours. It requires that in each proper non-empty subset of vertices S there are at most $|S| - 1$ edges between them. Indeed, having a subset of vertices with $|S|$ edges if $S \neq V$ means that there are subtours in the solution, for instance if $S = \{1, 2, 3\}$ as in Fig. 4b then $\sum_{i,j\in S: i\neq j} x_{i,j} = 3 > |S| - 1$.

3

The problem with model in Fig.5 is that there are exponential number of subsets S, e.g. $n = 10$ results in $2^{n-2} = 1022$ constraints of (16).

However, we may generate Constraint (16) in a lazy manner. In the beginning, the problem is formulated first without these constraints, i.e. only with Constraints (14) and (15). Then each time when the new integer solution is found during branching, we find a cycle in this solution; let S be the nodes in this cycle. If $|S| < n$, i.e. the solution contains a subtour, the corresponding constraint (16) is added to the model. Adding this constraint excludes this solution from the search space in the further run. The constraints are added to the model until the solution is the cycle of length n.

1.2.1 Lazy Constraints in Gurobi

Modern solvers, including Gurobi optimizer, allow the user to apply the lazy constraints trick by using so called *callbacks* that are called whenever a feasible solution is found.

1.2.1.1 Java
Lazy constraints have to be enabled by setting model parameter `GRB.IntParam.LazyConstraints` to 1.

```
GRBModel model = new GRBModel(env);
model.set(GRB.IntParam.LazyConstraints, 1);
```

To define a callback, inherit from class `GRBCallback` and override method `callback()`. The callback is then passed to model by `setCallback()`.

```
public class MyCallback extends GRBCallback {
  protected void callback() {
    // Callback is called when some event occur. The type of event is
    // distinguished using variable ''where'' defined in parent class.
    // In this case, we want to perform something when an integer
    // solution is found, which corresponds to ''GRB.CB_MIPSOL''.
    if (where == GRB.CB_MIPSOL) {
      // TODO: your code here...

      // Get the value of variable x[i, j] from the solution.
      // There are also methods for getting multiple values at once, see docs.
      double value = getSolution(x[i, j]);

      // Add lazy constraint to model.
      addLazy(...);
    }
  }
}

model.setCallback(new MyCallback());
model.optimize();
```

1.2.1.2 Python
Lazy constraints have to be enabled by setting model parameter `lazyConstraints` to 1.

```
model = g.Model()
model.Params.lazyConstraints = 1
```

To define a callback, create a function that accepts two arguments: `model` and `where`. The callback is then passed to model when calling `optimize()`.

```
def my_callback(model, where):
    # Callback is called when some event occur. The type of event is
    # distinguished using argument ''where''.
    # In this case, we want to perform something when an integer
    # solution is found, which corresponds to ''GRB.Callback.MIPSOL''.
    if where == GRB.Callback.MIPSOL:
        # TODO: your code here...
```

```
        # Get the value of variable x[i, j] from the solution.
        # You may also pass a list of variables to the method.
        value = model.cbGetSolution(x[i, j])

        # Add lazy constraint to model.
        model.cbLazy(...)

    model.optimize(my_callback)
```

1.2.1.3 C++ Lazy constraints have to be enabled by setting model parameter
`GRB.IntParam.LazyConstraints` to 1.

```
  GRBModel model(env);
  model.set(GRB_IntParam_LazyConstraints, 1);
```

To define a callback, inherit from class `GRBCallback` and override method `callback()`. The
callback is then passed to model by `setCallback()`.

```
  class MyCallback : public GRBCallback {
    protected:
      void callback() {
        // Callback is called when some event occur. The type of event is
        // distinguished using variable ''where'' defined in parent class.
        // In this case, we want to perform something when an integer
        // solution is found, which corresponds to ''GRB_CB_MIPSOL''.
        if (where == GRB_CB_MIPSOL) {
          // TODO: your code here...

          // Get the value of variable x[i, j] from the solution.
          // There are also methods for getting multiple values at once, see docs.
          double value = getSolution(x[i, j]);

          // Add lazy constraint to model.
          addLazy(...);
        }
      }
  }

  MyCallback cb;
  model.setCallback(&cb);
  model.optimize();
```

2 A homework assignment - image unshredding

Image unshredding is a problem of reconstructing shredded images from a set of stripes as closely
as possible to the original image. For example, Fig. 6 shows the shredded image and Fig. 7 is its
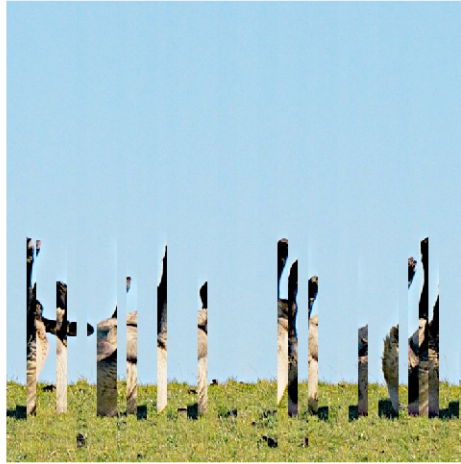reconstruction.

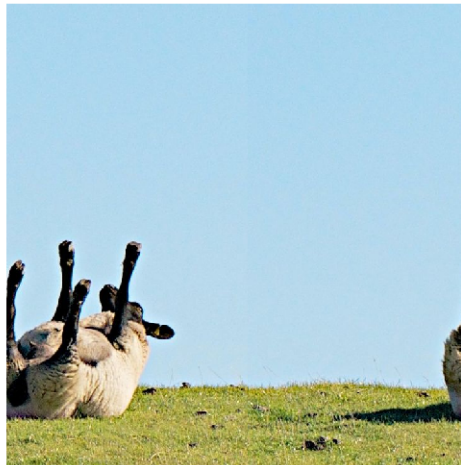Figure 6: Example of shredded image.

Figure 7: Example of reconstructed image.

Interestingly, quite reliable image reconstructions can be obtained by transforming this problem to Shortest Hamiltonian Path Problem (SHPP), where each stripe will represent one node in a graph. The idea is to order the stripes in such a way that adjacent stripes are "similar". For this, we need to define a distance function between two stripes.

Let w denote the width, i.e., the number of pixel columns in each stripe and let m denote the height, i.e., the number of rows in each stripe. Assume that we have two stripes $S^{(1)}$ (see Fig. 8) and $S^{(2)}$ (see Fig. 9). Both stripes have width $w = 2$ and height $h = 4$.
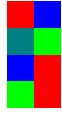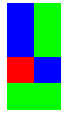
(a) Color image of the stripe.

$$\begin{bmatrix} 255 & 0 & 0 \\ 0 & 128 & 128 \\ 0 & 0 & 255 \\ 0 & 255 & 0 \end{bmatrix} \quad \begin{bmatrix} 0 & 0 & 255 \\ 0 & 255 & 0 \\ 255 & 0 & 0 \\ 255 & 0 & 0 \end{bmatrix}$$

(b) RGB values of the pixels in left and right column.

Figure 8: Stripe $S^{(1)}$.

(a) Color image of the stripe.

$$\begin{bmatrix} 0 & 0 & 255 \\ 0 & 0 & 255 \\ 255 & 0 & 0 \\ 0 & 255 & 0 \end{bmatrix} \quad \begin{bmatrix} 0 & 255 & 0 \\ 0 & 255 & 0 \\ 0 & 0 & 255 \\ 0 & 255 & 0 \end{bmatrix}$$

(b) RGB values of the pixels in left and right column.

Figure 9: Stripe $S^{(2)}$.

The distance function is a sum of the absolute difference between the RGB color components of the adjacent pixel columns of the stripes, i.e.,

$$\text{dist}(S^{(1)}, S^{(2)}) = \sum_{i=1}^{h} \sum_{c=1}^{3} |s_{i,w,c}^{(1)} - s_{i,1,c}^{(2)}| \tag{18}$$

Notice that the distance function is generally not symmetric since different columns are compared, i.e.

$$\text{dist}(S^{(1)}, S^{(2)}) = 1020 \tag{19}$$

$$\text{dist}(S^{(2)}, S^{(1)}) = 765 \tag{20}$$

Therefore, the image unshredding problem can be solved by computing the distances between every pair of stripes and solving the corresponding SHPP. The solution of the SHPP then represents the order of the stripes.

Moreover, we can further transform SHPP to TSP by adding a "dummy" node that has a zero distance to all other nodes (the nodes that will be connected to the dummy node in the TSP solution are the left and right edges of the reconstructed image.). The motivation for this transformation is that there exists really good solvers for TSP, whereas for SHPP this is not the case.

> **A homework assignment:** Implement a program that solves the image unshredding problem using TSP. Implement exact TSP solver using **lazy constraints**. Upload your source code to the CourseWare Upload System where it will be automatically evaluated.

> **Hint 1 (for Python):** Computing the distances between stripes in Python may lead to time-outs if not implemented efficiently. We strongly recommend using `numpy`, which is an optimized library for working with matrices using Matlab-like interface. Eq. (18) can then be computed using vectorized notation instead of two nested for-loops.

> **Hint 2:** Use the following interface to your TSP solver: `int[] solveTsp(double[][] distances)`, where `distances[i][j]` is a distance from node i to node j. The result of the function is an ordered list of node indices forming the optimal circuit.

> **Hint 3:** When an integer solution with multiple subtours is found in the callback, add the constraint that forbids the shortest one. Experimentally, forbidding shortest subtours shows good results.

2.1   Input and Output Format

Your program will be called with two arguments: the first one is absolute path to input file and the second one is the absolute path to output file (the output file has to be created by your program).

The input file has the following form

$$
\begin{array}{cccccccc}
n & w & h \\
s_{1,1,1}^{(1)} & s_{1,1,2}^{(1)} & s_{1,1,3}^{(1)} & s_{1,2,1}^{(1)} & \cdots & s_{1,w,3}^{(1)} & s_{2,1,1}^{(1)} & \cdots & s_{h,w,3}^{(1)} \\
s_{1,1,1}^{(2)} & s_{1,1,2}^{(2)} & s_{1,1,3}^{(2)} & s_{1,2,1}^{(2)} & \cdots & s_{1,w,3}^{(2)} & s_{2,1,1}^{(2)} & \cdots & s_{h,w,3}^{(2)} \\
\vdots \\
s_{1,1,1}^{(n)} & s_{1,1,2}^{(n)} & s_{1,1,3}^{(n)} & s_{1,2,1}^{(n)} & \cdots & s_{1,w,3}^{(n)} & s_{2,1,1}^{(n)} & \cdots & s_{h,w,3}^{(n)}
\end{array}
$$

One space is used as a separator between values on one line. All the values in the input file are non-negative integers. Values $s_{i,j,c}^{(k)} \in \{0, 1, \ldots, 255\}$ represent the color components of the pixels in stripes.

The output file has the following form

$$
\pi(1) \quad \pi(2) \quad \ldots \quad \pi(n)
$$

where π is an optimal permutation of the stripes (without the dummy stripe). All the values in the input file are positive integers.

Example 1

Notice that the first stripes correspond to stripes $S^{(1)}$ and $S^{(2)}$ from Fig. 8 and Fig. 9, respectively.

Input:

```
3 2 4
255 0 0 0 0 255 0 128 128 0 255 0 0 0 255 255 0 0 0 255 0 255 0 0
0 0 255 0 255 0 0 0 255 0 255 0 255 0 0 0 0 255 0 255 0 0 255 0
0 0 255 0 0 255 0 0 255 0 0 255 0 0 255 0 0 255 0 0 255 0 0 255
```

Output:

```
3 2 1
```

References

[1] D. L. Applegate, R. E. Bixby, V. Chvatal, and W. J. Cook, *The traveling salesman problem: a computational study.* The traveling salesman problem: a computational study, 2011.