Professor Dr. Stefan Hougardy
Fine Foos, M. Sc.

# Programming Exercise 2

**Task:** Implement a Branch-and-Bound algorithm for the Traveling Salesman Problem based on the Held-Karp lower bound.

**Algorithm:** Let $(K_n, c)$ be given, and consider a vector $\lambda \in \mathbb{R}^n$. Define edge costs $c_\lambda$ by $c_\lambda(\{i,j\}) := c(\{i,j\}) + \lambda(i) + \lambda(j)$. Given a 1-tree $T$, set

$$c(T, \lambda) := c(T) + \sum_{v \in V(T)} (|\delta_T(v)| - 2) \cdot \lambda(v).$$

Now, let $T_\lambda$ be a min-$c_\lambda$-cost 1-tree (which also minimizes $c(T_\lambda, \lambda)$), and set

$$HK(K_n, c, \lambda) := c(T_\lambda, \lambda).$$

Then $HK(K_n, c, \lambda)$ is a lower bound on the minimum weight of any tour in $(K_n, c)$, so $HK(K_n, c) := \max_{\lambda \in \mathbb{R}^n} HK(K_n, c, \lambda)$ is also a lower bound on the minimum weight of a tour. In particular, if $T_\lambda$ is 2-regular, it is an optimum tour. The idea of the algorithm is to use Branch-and-Bound to compute a 2-regular 1-tree of minimum cost.

It has been shown in the lecture that we can compute a vector $\lambda$ that approximately maximizes $HK(K_n, c, \lambda)$ using an iterative combinatorial algorithm, c.f. below.

In the Branch-and-Bound algorithm, we maintain an upper bound $U$ on the cost of a shortest tour, and a set $Q$ of branching nodes to be processed. Each node in the Branch-and-Bound tree is represented by two disjoint sets of edges $(R, F)$ which are initially empty, i.e., start with $Q = \{(\emptyset, \emptyset)\}$. The node $(R, F)$ represents all tours in $K_n$ where all edges in $R$ are required, and all edges in $F$ are forbidden. Clearly the node $(\emptyset, \emptyset)$ represents all tours in $K_n$.
Now, while $Q$ is not empty, select a node $(R, F) \in Q$ and set $Q := Q \setminus \{(R, F)\}$. Compute $\lambda$ s.t. the weight of a min-$c_\lambda$-cost 1-tree $T$ with $R \subseteq E(T) \subseteq E(K_n) \setminus F$ is approximately maximum and let $T$ be such a 1-tree. If $HK(K_n, c, R, F) := c(T, \lambda) \geq U$, we know that the node $(R, F)$ does not represent a solution leading to better cost than $U$, so we can discard $(R, F)$. So assume $c(T, \lambda) < U$. If $T$ is 2-regular, it is an optimum tour among those represented by $(R, F)$, so we can update $U$. Otherwise, there needs to be a vertex $2 \leq i \leq n$ with $|\delta_T(i)| > 2$. We can assume that $|\delta_T(i) \setminus (R \cup F)| \geq 2$,

because whenever two edges incident to a vertex $v$ are required, we can forbid all other edges incident to $v$. Let $e_1, e_2 \in \delta_T(i) \setminus (R \cup F)$ be two distinct edges in $T$ incident to $i$ that we have not yet branched on. Partition $(R, F)$ into the three branching nodes

$$\begin{array}{ll} (R, & F \cup \{e_1\}). \\ (R \cup \{e_1\}, & F \cup \{e_2\}), \\ (R \cup \{e_1, e_2\}, & F), \end{array}$$

where the last node is omitted if there is already a required edge incident to $i$, and add these nodes to $Q$.
When $Q$ is empty, we know that $U$ equals the cost of a shortest tour in $(K_n, c)$, and we can return the tour that led to the current value of $U$.

**Usage:** Your program should be called as follows:

```
<program_name> <file.tsp> [<tour.tsp>],
```

where the second argument is optional.

**Input:** The argument `file.tsp` is mandatory (i.e., your program should exit with an error message if it is not present), and it encodes the graph $(K_n, c)$ for which your program should find a shortest tour. It is expected to be in TSPLIB format.

In case the second argument is given, the computed tour should be written to a file of this name.

The TSPLIB format allows to encode arbitrary instances, however, we restrict to instances with rounded Euclidean costs, i.e. distances are computed by taking the nearest integer to the Euclidean distance (rounded up in case of $x.5$). If the input is not correct, your program should exit with an error message (i.e. in particular, throw no exceptions).

The TSPLIB format works as follows: First, there is a *specification* section, encoding data in the format `<keyword>: <value>`. The only keyword relevant for us is `DIMENSION`, which specifies the number of vertices. Other possible keywords are `NAME`, `TYPE`, `COMMENT` and `EDGE_WEIGHT_TYPE` (which we ignore because we always use rounded Euclidean costs).

Then, one lines says `NODE_COORD_SECTION`, and after that, lines are of the form `i x y`, where `i` is an integer specifying the node index and `x` and `y` are reals specifying the position of the $i$-th node. Note that indices start at 1.

The input may be ended by a line reading `EOF`, but this is optional.

**Output:** Your program should write the cost of a shortest tour in $(K_n, c)$ to `stdout`. No other output may be written to `stdout` / `std::cout`. Furthermore, if the optional argument `tour.tsp` is specified, your program should write an encoding of the found optimum solution to the specified file in TSPLIB format.

A tour is encoded as follows in TSPLIB format: The encoding starts with a header of the form

```
TYPE : TOUR
DIMENSION : n
TOUR_SECTION
```

where $n$ is the number of vertices in the instance. Now, $n$ lines follow, each consisting of a single integer $i$ with $1 \leq i \leq n$. These lines specify the order in which the vertices are visited in an optimum tour. Finally, the encoding ends with these two lines:

```
-1
EOF
```

**Implementation:** In [1], an effective implementation of the algorithm is described, which may be helpful (see below). It is recommended to follow [1] as close as possible for at least the lower bound computation. They also describe heuristics to find better values of $U$ during the algorithm, which you do not need to implement but may speed up the computation. Also, you may start with $U = \infty$, although using a heuristic to determine a better value of $U$ before the algorithm may lead to better results.

**Branching:** During branching, there are several choices to be made that will influence the runtime of the algorithm. Most importantly, it is not specified how to select the next element $(R, F) \in Q$ to be processed. The two most prominent choices are *best-bound* and *depth-first.*

Best-bound always selects a node $(R, F)$ minimizing $HK(K_n, c, R, F)$, i.e., $Q$ is a priority queue. This approach processes a minimum amount of nodes, but may take a long time to find good values of $U$. Note that $HK(K_n, c, R, F)$ needs to be evaluated *before* inserting a node $(R, F)$ into $Q$, and insertion is only performed if the computed 1-tree is not 2-regular and $HK(K_n, c, R, F) < U$.
Depth-first always processes an element of $Q$ of maximum depth in the branching tree, i.e., $Q$ is a stack. This approach tends to quickly find feasible solutions and allows to delay the evaluation of $HK(K_n, c, R, F)$, but

may enumerate more branching nodes.
We recommend to use best-bound. Other choices to be made are the selection of the vertex $i$ to branch on, and the selection of the edges $e_1$ and $e_2$.

**Lower bound:** In the lecture, it has been shown that we can find a vector $\lambda$ that approximately maximizes $HK(K_n, c, \lambda)$ by starting with $\lambda_0 \equiv 0$ and repeating the following step. In iteration $i \geq 0$, compute a min-$c_{\lambda_i}$-weight 1-tree $T_i$ and set

$$\lambda_{i+1}(x) := \lambda_i(x) + t_i \cdot (|\delta_{T_i}(x)| - 2), \tag{1}$$

where the sequence $t_i$ converges to 0 and satisfies $\sum_{i=0}^{\infty} t_i = \infty$, e.g., $t_i := \frac{1}{i+1}$. In practice, a good choice of the step lengths $t_i$ is crucial for the algorithm to compute a good approximation within a bounded number of iterations. In particular, the step lengths of course should depend on the costs $c$. In [1], it is proposed to fix the number of iterations $N$ and the initial step length $t_0$ in advance, and then letting the $t_i$ converge to 0 (e.g., $t_N = 0$) with a constant second order difference

$$(t_i - t_{i+1}) - (t_{i+1} - t_{i+2}) \equiv \text{const}$$

under the (arbitrary) condition

$$(t_0 - t_1) = 3(t_{N-1} - t_N).$$

This can be achieved by defining $\Delta_0 = \frac{3t_0}{2N}$ and $\Delta^* = \frac{t_0}{N^2-N}$, and, for $0 \leq i \leq N-1$,

$$\begin{aligned} t_{i+1} &= t_i - \Delta_i, \\ \Delta_{i+1} &= \Delta_i - \Delta^*. \end{aligned}$$

It remains to choose $N$ and $t_0$. Since the computations for branching nodes $(R, F)$ that are not the root of the branching tree (i.e., $(\emptyset, \emptyset)$) can (and should!) start with the vector $\lambda$ of their parent, it is reasonable to use more iterations at the root node. Hence, [1] propose to use $N = \lceil \frac{n^2}{50} \rceil + n + 15$ at the root node and $N = \lceil \frac{n}{4} \rceil + 5$ at other nodes. To let $\lambda$ scale with $c$, at the root node set $t_0 = \frac{c(T_0)}{2n}$, where $T_0$ is the 1-tree computed in the first iteration, i.e., a minimum cost 1-tree of $(K_n, c)$. At other nodes use $t_0 = \frac{1}{2n}\sum_{i=1}^{n} |\lambda_{\text{root}}(i)|$, where $\lambda_{\text{root}}$ is the vector $\lambda$ computed at the root node.
Furthermore, Volgenant and Jonker [1] observed oscillating degrees during the iterations. To dampen these oscillations, for $i \geq 1$, they replace (1) by

$$\lambda_{i+1}(x) = \lambda_i(x) + t_i \cdot \Big(d \cdot (|\delta_{T_i}(x)| - 2) + (1-d) \cdot (|\delta_{T_{i-1}}(x)| - 2)\Big)$$

with $d = 0.6$.
A minimum weight 1-tree can be obtained by starting with a MST on $\{2, \dots, n\}$ and adding the two cheapest edges incident to vertex 1. This approach can be extended to computing a minimum weight $(R, F)$-1-tree, i.e., a 1-tree $T$ of minimum weight s.t. $R \subseteq E(T) \subseteq E(K_n) \setminus F$: Artificially modify the costs of edges in $F \cup R$ s.t. all edges in $R$ are chosen and no edges in $F$ are chosen. More precisely, set the costs of edges in $F$ to $\infty$, and set the costs of edges $R$ to a value strictly smaller than the cost of any non-required edge. Of course, the weight of $T$ needs to be evaluated w.r.t. the original costs.

**Language:** Your program should be written in C or C++, although the use of C++ is strongly encouraged. Your program will be compiled with gcc 9.3.0 and C++17 using `-pedantic -Wall -Wextra -Werror`, i.e., all warnings are enabled and each remaining warning will lead to compilation failure. Your submission must contain a bash script `compile.sh` which compiles your program (which for example may just call gcc making use of the mandatory compile options listed above). Program evaluation will be performed on Linux. The standard library can be used as you wish, but no other libraries.

**Evaluation:** Make sure to implement core steps of the algorithm as fast as possible. Compute min-weight 1-trees in $\mathcal{O}(n^2)$ time. Note that e.g. using Kruskal's algorithm with comparison-based sorting leads to a runtime of $\mathcal{O}(n^2 \log n)$ on complete graphs. Your algorithm should be able to solve each of the instances `berlin52.tsp`, `st70.tsp`, `eil76.tsp`, `rat99.tsp`, `rd100.tsp`, `eil101.tsp` and `lin105.tsp` in at most a few seconds, more points are given for solving larger instances fast as well.

The elegance, cleanness and organization of your code will be evaluated. Make sure to add good, non-redundant documentation and give the variables, functions, types and so on meaningful names that make their role clear. Break your complicated functions into small simple ones, break your program into a few units etc. Of course, your program may not trigger undefined behavior. In particular, your program must be `valgrind`-clean, i.e., must not leak memory and must not perform invalid operations on memory.

**Help:** On the eCampus website you will find some instances with known optima that you can use to test your program.

**Please submit your programs in groups of 2 or 3 students.** (25 points)

**Deadline:** January 18, 23:55. The websites for lecture and exercises can be found at:

`https://ecampus.uni-bonn.de/goto.php?target=crs_2282562`

In case of any questions feel free to contact me at foos@or.uni-bonn.de.

## References

[1] Volgenant, Ton, and Jonker, Roy, *A branch and bound algorithm for the symmetric traveling salesman problem based on the 1-tree relaxation,* European Journal of Operational Research, 1982.