Graduate Program in Computer Science, Universidade Federal de Santa Catarina
INE410104 (Design and Analysis of Algorithms)

Solutions to the 2nd Set of Exercises
Emmanuel Podestá Jr., Gustavo Zambonin, Matheus S. P. Bittencourt

1 Answers

The source code to the programming exercises is available here.

1. (a) We refer to the file `optimization/cheap_route.py` in the repository above.

   (b) The description of the directed edge-weighted input graph $G = (V, A, w)$ considers the function $w : V \to \mathbb{R}^+$ to output values in kilometres. By making use of the other parameters, namely a function $p : V \to \mathbb{R}_*^+$ that describes toll roads, the gas price $g \in \mathbb{R}_*^+$ in currency per litre, and the vehicle autonomy $a \in \mathbb{R}_*^+$ in kilometres per litre, we create a function $w' : V \to \mathbb{R}^+$ with values in the currency unit:

   $$\forall v \in V,\ w'(v) = \begin{cases} \frac{g}{a} \cdot w(v) + p(v) & \text{if } w(v) \neq 0, \\ 0 & \text{if } w(v) = 0. \end{cases}$$

   It is known that Dijkstra's algorithm returns the shortest path between two vertices in a graph. Applying this algorithm on the source and destination nodes $s, t \in V$ and a graph $G' = (V, A, w')$, we obtain the lowest cost route from s to t on G'.

2. (a) We refer to the file `optimization/spanning_tree.py` in the repository above.

   (b) The strategy implemented makes use of Kruskal's algorithm for finding the minimum spanning forest. Thus, the complexity of the code is quasilinear in the number of edges, *i.e.* in asymptotic notation, it is $\mathcal{O}(E \lg E)$. Additionally, the sorting procedure used inside Kruskal to sort the edges is also quasilinear, due to Python's Timsort algorithm. To obtain the edges with degree higher than three, the procedure needs only to look at every edge, thus being linear in E and of lower complexity than Kruskal.

3. (a) We refer to the file `optimization/multiple_knapsack.py` in the repository above.

   (b) The proposed solution is not an optimal one, it uses dynamic programming to solve each "knapsack", *i.e.* the trucks, individually, starting with the truck that has the smallest capacity. Since the dynamic programming solution does not optimally solve the knapsack problem, and the trucks were inspected in a greedy manner rather than globally, our solution to the "multiple knapsack" problem as proposed is not optimal either. However, this strategy can be pretty effective. It combines an efficient solution for a single knapsack, and a greedy approach to the fact of having multiple knapsacks. It achieves up to 71.2% of the optimal profit in average on our known tests. Additionally, a Fortran interface to `MTM.FOR` from this collection of knapsack-related implementations

is given in `optimization/mkp_wrapper.f90`. It calculates the optimal solution for the multiple knapsack problem when there are many more items than knapsacks.

4. (a) A decision problem $p \in \mathbf{NP}$ is called **NP**-complete if every decision problem in **NP** has a polynomial-time, many-one reduction to p. The travelling salesman problem in its decision version (TSPd) is an example of a **NP**-complete problem. Therefore, if one were to give a polynomial time algorithm to solve TSPd, and considering that it is possible to reduce all problems in **NP** to an instance of TSPd, one could solve these in polynomial time. In other words, **P** would be equal to **NP**.

   (b) Recall that a problem p, not necessarily in **NP**, is **NP**-hard if there exist polynomial-time, many-one reductions from all problems in **NP** to p. Usually, it is said that **NP**-hard problems are "at least as hard as the hardest problems in **NP**", *i.e.* those that are **NP**-complete. If a polynomial reduction from a problem p to q exists, then by the definition above, q is in the hardest class of problems, and may not even be decidable.

   (c) Solving and verifying a problem in a quantity of time polynomial to the size of the input are, respectively, the informal definitions to the complexity classes **P** and **NP**. Problems in the latter class have no known polynomial time algorithms to solve them, but given a possible solution, there exists a polynomial time algorithm that can decide if it is valid or not. For example, factoring a large integer is known to be computationally hard, but given its factors, it is trivial to check for a correct answer.

   (d) Approximation algorithms are probably bounded to find solutions within some "distance" of solutions to the original problems. This is equivalent to the idea that the approximate solution returned is moderately good, or no worse than the worst case scenario. To the contrary, heuristics present no mathematical proofs that constrain the quality of the solution, often requiring trade-offs to work efficiently or accurately.