185.A05 Advanced Functional Programming SS 21

Monday, 22 March 2021

Assignment 2

on Chapter 3 and Chapter 8

Topics: Algorithm Patterns (Backtracking, Greedy Search, Dynamic Programming), Abstract Data Types (Stacks, Priority Queues, Tables)

Submission deadline: Monday, 12 April 2021, 12am

Regarding the deadline for the second submission: Please, refer to „Hinweise zu Organisation und Ablauf der Übung“ available at the homepage of the course.

Important:

1. Carefully read and follow the instructions outlined in the complementary files provided with assignment 0. If you have any questions regarding them, ask your questions in the TISS forum. Following these instructions is essential to ensure a smooth processing of your submitted file with the test system.

2. Store all functions to be written for this assignment in a top-level file named

   `Assignment2.hs`

   of your group directory. Take advantage of the template file `Assignment2.hs` and extend it as required. Comment your program meaningfully, and use auxiliary functions and constants, where reasonable. The very same file name shall be used for the second submission of assignment 2.

3. *Do not use self-defined modules!* If you want to re-use functions (written for other assignments), copy them to `Assignment2.hs`. Import declarations for self-defined modules will fail: Only `Assignment2.hs` will be copied for the (semi-automatical) evaluation procedure, no other ones.

4. Your programs will be (semi-automatically) evaluated on `g0` using the there installed GHC interpreter of GHC version 8.65. If you use a different tool or a different version of GHC for program development, please, double-check well in time before the submission deadline that your programs behave on `g0` and the there installed GHC interpreter version as you expect.

Programming assignments:

We consider a variant of the travelling salesman problem. In our variant of the problem, the vaccine coordinator of the Austrian federal government performs round trip visits of all or some of the Austrian federal state capitals in order to discuss challenges of the ongoing vaccination campaign with local authorities. In order to minimize travel time and costs the state capitals visited on a round trip shall be visited exactly once in an order that miminizes the overall travel distance. To this

end we assume that the state capitals are connected by a system of one-way roads, however, we do not assume that the state capitals are pairwise linked by direct one-way roads. There may be a one-way road from state capital A to state capital B but not vice versa; and there may be state capitals, which are not linked at all by a connecting one-way road. Each round trip shall start and end in the state capital coming lexicographically first in the list of state capitals to be visited on a round trip. Finally, we assume that the vaccine coordinator travels with a battery powered e-car with limited range. In order to ensure to not run out of power on the road, the range of the car sets an upper limit on the travel distance of round trips that can be chosen by the vaccine coordinator.

We model the travelling vaccine coordinator problem as follows:

```
type Nat1      = Int
type Kilometer = Nat1
type Distance  = Kilometer
type MaxRange  = Kilometer

data Capital = Bregenz | Eisenstadt | Graz | Innsbruck | Klagenfurt
             | Linz | Salzburg | StPoelten | Wien
             deriving (Eq,Ord,Enum,Show)

type From    = Capital
type To      = Capital
type Roadmap = From -> To -> Maybe Distance
 -- Just values of Maybe Distance denote the length of the
 -- one-way road from capital 'from' to capital 'to'. The Nothing
 -- value indicates the absence of such a one-way road.

type ToBeVisited = Capital -> Bool
 -- A capital is to be visited on a round trip iff it is mapped to True.

type RoundTrip = [Capital]
 -- Each round trip starts and ends in the state capital coming lexicographi-
 -- cally first, e.g. [Eisenstadt,Wien,Linz,Salzburg,Innsbruck,Eisenstadt]
type RoundTripTravelDistance = Kilometer

type Itinerary = (RoundTrip,RoundTripTravelDistance)
```

1. Use the algorithm pattern for *backtracking search* of Chapter 3 to implement:

   ```
   vcp1 :: Roadmap -> ToBeVisited -> MaxRange -> Maybe [Itinerary]
   ```

   which yields as `Just` value all round trips not exceeding the maximum range of the vaccine coordinator's e-car. If there is more than one, they can be listed in any particular order. If there is none, `vcp1` yieds the value `Nothing`.

2. Modify the algorithm pattern for *backtracking search* of Chapter 3 to terminate after the first solution has been discovered. Use the modified *backtracking* algorithm pattern to implement:

   ```
   vcp2 :: Roadmap -> ToBeVisited -> MaxRange -> Maybe Itinerary
   ```

   which behaves like `vcp1` but returns only one round trip proposal, if there is one (or more), otherwise it returns `Nothing`.

3. Use the algorithm pattern for *greedy search* of Chapter 3 to implement:

   ```
   vcp3 :: Roadmap -> ToBeVisited -> MaxRange -> Maybe Itinerary
   ```

   Like `vcp2`, `vcp3` returns a round trip proposal, if there is one (or more) but the one returned if some exist(s), is the one uniquely determined by the principle of greedy search, where uniqueness is ensured as follows: as we are interested in round trips as short as possible, greedy search picks that capital as the next one on a trip which is nearest to the current one. If two capitals are equally far away from the current one, it (arbitrarily) picks the one coming alphabetically first in order to ensure uniqueness. If a valid round trip does not exist, `vcp3` returns `Nothing`.

4. **Without submission:**
   (a) Test and validate your implementations with test data of your own choice.
   (b) Use the result of `vcp1` to double-check, if
      i. `vcp2` yields a round trip with minimum overall travel distance, if there is one. If not, why not?
      ii. `vcc3` does so? If not, why not?
   (c) What is the asymptotic complexity ('big-O') of `vcp1`, `vcp2`, `vcp3`?

Dynamic programming, a historical mis-nomer for a computing approach (called programming) storing results of computations for later reuse in a table instead of recomputing them, often helps in overcoming the complexity of brute force approaches for solving optimization problems with an exponential number of solution candidates.

Informally, dynamic programming succeeds if a problem can be decomposed into smaller problems such that the optimal solution of the original problem can be computed from the optimal solutions of the smaller ones.

This is formalized in Bellman's optimality criterion, which is thanks to and named in honor of Richard E. Bellman:

Bellman's Optimality Criterion
An optimization problem P satifies Bellman's Optimality Criterion, if:

(i) Every (non trivial) instance I of P can be decomposed into m smaller and independent of each other instances I_k, $1 \leq k \leq m$, of P such that

(ii) every combination of arbitrary optimal solutions of $I_1, \ldots, I_k$ yields an optimal solution of I.

Finding the *longest common subsequence* (*LCS*) of two strings is an example of an optimization problem satisfying Bellman's optimality criterion. The definition of this problem reuires the following notion: If s is a string, a *substring* of s can be constructed by deleting arbitrarily many (not necessarily neighboured) characters of s. The LCS problem is then the following:

- Let $a_1 \ldots a_m$ and $b_1 \ldots b_n$ two (possibly empty) strings. Find the length of the longest common substring(s) of $a_1 \ldots a_m$ and $b_1 \ldots b_n$.

For example, given the two strings `"BANANAS"` and `ANALYSIS`, their longest common substring is `"ANAS"` of length 4, and of `"BANANAS-FARMS"` and `"BIDIRECTIONAL-ANALYSIS"` it is the substring `"BNANASS"` of length 7.

The following recurrence relation shows how the optimal solution of an instance I of LCS can be computed from optimal solutions of smaller instances of LCS I can be decomposed into. Let $d(i, j)$ denote the length of the longest common substring(s) of $a_1 \ldots a_i$ and $b_1 \ldots b_j$. Then $d(i, j)$, $0 \leq i \leq m$, $0 \leq j \leq n$, enjoys the recurrence relation:

$$d(i,j) \;=\; \begin{cases} 0 & \text{if } i = 0 \vee j = 0 \\ max\left(d(i,j-1), d(i-1,j), d(i-1,j-1) + \left\{ \begin{array}{ll} 1 & \text{if } a_i = b_j \\ 0 & \text{if } a_i \neq b_j \end{array} \right\}\right) & \text{if } i,j > 0 \end{cases}$$

5. Use the algorithm pattern for *dynamic programming* of Chapter 3 to implement:

   ```
   lcs :: String -> String -> Int
   ```

6. **Without submission:**

   (a) Test and validate your implementation of `lcs` with test data of your own choice.

   (b) What is the asymptotic complexity ('big-O') of `lcs` compared to a brute force approach for solving the LCS problem?

   (c) Memoization is often (considered) an immediate alternative to dynamic programming. Does this apply to the LCS problem, too? Explain your reasoning.

Iucundi acti labores.
Getane Arbeiten sind angenehm.

Cicero (106 - 43 v.Chr.)
röm. Staatsmann und Schriftsteller

Note: The deadlines for the first submission of assignment 2 and the second submission of assignment 1 are both on Monday, 12 April 2021, at noon.