Functional and Logical Programming (PFL) Project 1

Group T06_G09

Name	E-mail	Contribution
Bruno Ricardo Soares Pereira de Sousa Oliveira	up202208700@up.pt	50%
Rodrigo Albergaria Coelho e Silva	up202205188@up.pt	50%

Tasks Developed

- Bruno Ricardo Soares Pereira de Sousa Oliveira
  - Implementation of the Bitmask and Heap structures
  - Development of main and auxiliary functions
  - Test of the functions with properties and descriptions
  - Documentation of the code
- Rodrigo Albergaria Coelho e Silva
  - Implementation of the Set and Map structures
  - Development of main and auxiliary functions
  - Test of the functions with properties and descriptions
  - Documentation of the code

Algorithm Explanations

Dijkstra's algorithm: `shortestPath`

The `shortestPath` function computes all possible shortest paths between a start and end city in the roadmap. The approach that we used for this problem, also known as "Single Pair Shortest Paths", was based on the Dijkstra's algorithm.

The main idea of this algorithm is to start with a root node and sequentially explore the unvisited node whose path is closest to the start. By doing this, we will explore all the shortest paths connecting the initial node to the other nodes.

Additionally, if for each node we keep track not only of the smallest distance to reach it but also of the path that was used to reach it, when the algorithm finishes (which is when all the nodes have been visited or there are no more edges to explore) we will have for each node the smallest distance between the starting node and that specific node, as well as the paths that can be used to reach it.

At this point, by retrieving this information for the end node which we were trying to reach, we will have all the shortest paths that connect the starting node to the end node (if there are any).

The Dijkstra's algorithm can be performed in `O(E log E)`, being faster for sparse graphs than dense graphs. Therefore, we should use a graph representation that is more efficient for sparse graphs, like an adjacency list. However, since list in Haskell are linked lists, retrieving the neighbors of a node in Haskell needs a linear search over the adjacency list (of length `V`), which must be done for each vertex, in the worst case. Therefore, this adjacent retrieval will take `O(V^2)` time, losing the efficiency for sparse graphs. To achieve a better time complexity, we decided to use an **adjacency map** representation, similar to the adjacency list, where each vertex and a list with its neighbors are stored in an entry of a map, allowing for logarithmic-time lookups. This way, we cut the time complexity of the algorithm from `O(E log E + V^2)` down to `O(E log E)`, which is optimal.

Additionally, since each shortest path will not pass through a vertex more than once, it will have a length of `V` at most, and reconstructing the paths thus takes `O(N * V)`, with `N` being the number of shortest paths between the two points specified. Depending on the graph family, this number can have a super-exponential growth with the number of vertices, but, for most cases, there will be only one path.

In addition, in order to implement this algorithm, we used: - **Map** to store the distance and paths of the shortest path between the start and each city. This allowed us to retrieve lookup the information of the shortest path for each city, during the algorithm, in logarithmic time complexity. This complexity is due to the way the map is implemented based on an AVL Tree set structure where the values are key-value entries comparable only by the key. - **Min Heap** to repeatedly retrieve the smallest path that ends in an unvisited node. This data structure allowed us to insert and pop the smallest element stored in logarithmic time complexity, improving the overall performance of the algorithm. Our binary heap implementation is based on a leftist tree, a type of binary tree that stores the rank (shortest distance to an null node) for each node, and ensures that the rank of the left branch is never smaller than the rank of the right branch. This means that the shortest path to a leaf node will always be the rightmost path, having a length that is, at most, logarithmic to the number of elements in the heap.

There are also some notes on our implementation we want to point out: - We took advantage of Haskell being lazy-evaluated to reconstruct the shortest paths. I.e., in each state of the algorithm, we define the new cities of the path and, in the end, we lookup only the shortest paths that end in the last vertex, and only those will be calculated. This is much simpler than storing the predecessors of each node and performing a DFS to restore the paths, which would be the only viable option on imperative, strictly-evaluated languages. - Since vertices are visited in increasing order of distance to the origin, after the destination vertex is visited, we can stop the algorithm if a state has a distance higher than the shortest distance to the destination. Although this does not improve the complexity of the algorithm, it can make the function run much faster for graphs where the shortest path is straightforward (e.g. the origin and the destination are connected by an unit weight edge).

One final note is that due to the way Dijkstra's algorithm works, it may not return the correct results (or even terminate) in case the graph contains negative edges or negative cycles. This is not due to our implementation, but rather due to the algorithm itself. For this project, since we are working with cities and roadmaps, we believe it is safe to assume all distances between cities are non negative.

Held-Karp's algorithm: `travelSales`

The `travelSales` functions aims to calculate the shortest hamiltonian circuit in a graph, that is, the shortest path that goes through all of the vertices only once and returns to the start. This problem is known as the Traveling Salesperson Problem (TSP) and to solve it efficiently, we used a dynamic programming approach, based on the Held-Karp algorithm.

This algorithm is based on the observation that the shortest path that goes through a set of vertices follows optimal substructure, i.e. the shortest path that goes from the origin `u` to a vertex `v`, through the vertices in a set `S` in some order (such that `u` and `v` are not in `S`), contains the shortest path that starts in `u` and passes through all the vertices in `S`. This way, we can solve the TSP by solving the following recurrence:

```
c[i, {}] = w[i, n]                                  , i != n
c[i, S]  = minimum [w[i, j] + c[j, S\{j}] | j <- S] , i != n, i not in S
```

where `c[i, S]` represents the cost of the shortest path that begins in `i` and goes through each vertex of `S` exactly once, in no particular order, and `n` is a node to be defined as the start and ending node of the cycle.

To implement this algorithm, we needed a way to access the distance between two cities in O(1) time, ideally. The algorithm also goes through each pair of vertices the same number of times, for each graph with a set number of vertices, so it does not benefit from a representation that benefits sparse graphs.

Using an **adjacency matrix** meets this requirement, since it represents the graph as a V x V matrix of the distances between each pair of vertices, which can be accessed in constant time.

In addition, to solve the recurrence efficiently, we also used, as auxiliary data structures: - **Array**, which is the basis of the adjacency matrix implementation, since it provides constant-time random access to any element. It is also used for the dynamic programming table, for the same reason. - **Bitmask**, a structure that represents a set of selected elements by setting the bits of an integer. We used this in the computation of the dynamic programming table, since it not only allows all the operations to test and set/clear bits in constant-time (using a fixed-size integer), but is also indexable, which a normal list or set is not and, therefore, cannot be used to index the table. One consideration to be made is that we used the `Int` type to represent the bitmask, so the maximum number of vertices is bound to the number of bits in the representation of `Int`. For 64-bit machines, this corresponds to 64 bits, which shouldn't be a limitation if the user has less than zettabytes of RAM, which the algorithm will need at those scales. - **Map**, not specifically in the algorithm, but in the overall function implementation we used a map to convert the cities to Int values during the creation of the adjacency matrix, as well as, the reverse operation to print the solution to the `travelSales` problem in the end. The implementation of this map is similar to the one mentioned above which uses an AVL Tree set of Key-Value entries, allowing for logarithmic insertions and lookups.

Implemented this way, the program can solve the TSP with time complexity `O(V^2*2^V)` and space complexity `O(V*2^V)`, with `V` being the number of vertices in the graph. Although the time complexity is exponential, it is much better than the super-exponential complexity of the brute-force/backtracking algorithm, which runs in `O(V!)` time.

Since the algorithm was implemented in Haskell, there are some aspects we can take advantage of: + In Haskell, arrays can be defined recursively, giving a very easy way to compute the problem efficiently by only writing the recurrence in the table definition. This also takes care of the order at which the table is calculated: due to lazy evaluation, Haskell will convert any bottom-up approach to dynamic programming into a top-down one, which means that, after defining the table, we only need to access the wanted element to solve the problem. + The table consists of an array of pairs `(distance, path)`, where `distance` is the cost of the shortest paths and `path` is the shortest path calculated. Since Haskell has lazy evaluation, we can define all the paths at each step in the table, and simple access the needed path at the end, and that one will be the only one calculated, just as we did in the `shortestPath` function.

Testing

For this project, we also used Quickcheck to develop tests for all of the data structures used based on their properties.

Besides the data structures we also implemented tests for all of the functions using both unit tests based on the expected results for the example graphs (gTest1, gTest2 and gTest3), as well as properties for each of them.

To test our function implementations and custom data structures, we set up a Cabal project, configured with QuickCheck and HSpec, a testing framework for Haskell that has integration with QuickCheck and allows to use a combination of various types of tests.

For almost every function, we defined unit tests, to test known values of the example graphs, and property-based tests, through QuickCheck, which generated automatic test cases to verify certain condition of our implementations. Some examples of properties we specified are that `cities` does not return duplicate cities and that `Set` always forms a balanced tree.

To generate property tests for functions that accept roadmaps and such, we also created generators that returned arbitrary values of types `City`, `(City,City,Distance)`, `Path` and `Roadmap`, with shrinking capabilities.

Some examples of the tests developed are shown below:

```
{- ... -}

newtype GoodRoadMap = GoodRoadMap RoadMap deriving (Show, Eq)

{- ... -}

instance Arbitrary GoodRoadMap where
  arbitrary = do
    rawMap <- listOf (arbitrary :: Gen GoodEdge)
    return $ GoodRoadMap $ removeDuplicateEdges $ map coerce rawMap

  shrink (GoodRoadMap roadMap) = [GoodRoadMap $ removeDuplicateEdges $
    map coerce edges | edges <- shrink $ map GoodEdge roadMap]

{- ... -}

prop_travelSalesSameEnds :: GoodRoadMap -> Property
prop_travelSalesSameEnds (GoodRoadMap roadMap) = let circuit = travelSales roadMap
  in not (null circuit) ==> head circuit == last circuit

prop_travelSalesPassesEachCityOnce :: GoodRoadMap -> Property
prop_travelSalesPassesEachCityOnce (GoodRoadMap roadMap) = let
  circuit = travelSales roadMap
  mapCities = cities roadMap
  in not (null circuit) ==> length circuit == length mapCities + 1 &&
    sortUnique circuit == mapCities

{- ... -}

main :: IO ()
main = hspec $ do

  describe "travelSales" $ do
    it "Determines correct circuit" $ do
      travelSales gTest1 `shouldSatisfy` sameCircuit
        ["0", "1", "2", "3", "4", "5", "6", "8", "7", "0"]
      travelSales gTest2 `shouldSatisfy` sameCircuit
        ["0", "1", "3", "2", "0"]
```

```haskell
    it "Checks if no hamiltonian circuit exists" $ do
      travelSales gTest3 `shouldBe` []

    prop "Circuit has the same ends" $ do
      prop_travelSalesSameEnds

    prop "Circuit passes through each city exactly once" $ do
      prop_travelSalesPassesEachCityOnce
```

References

1. Fethi Rabhi and Guy Lapalme. *Algorithms: a functional programming approach.* Addison-Wesley, 2nd edition, 1999.