Informatics 2 – Introduction to Algorithms and Data Structures

Lab Sheet 4: Greedy Algorithms in Python
(and some Dynamic Programming)

week 2, semester 2: 20th-24th January 2025

In this Lab Sheet, we will get some practice in writing programs to implement some of the Greedy Algorithms (and some basic dynamic programming) we are currently seeing in lectures. Most of the basic concepts you need to develop implementations have been introduced in earlier Labs - how to define arrays (including multi-dimensional arrays), print functionality, and timing-of execution using `timeit`.

The specific problems that we consider in this sheet are *Single-source shortest paths* (solved by different variants of Dijkstra's Algorithm) and *coin-changing* (addressed by direct recursion, then dynamic programming). Dijkstra's algorithm was covered in detail in lecture 16. Coin changing was introduced in lecture 17 and the dynamic programming algorithm will be shown in lecture 18 (on Tuesday 22nd January) - however, it should be possible to work on the implementation of Dyn-Coins at the end of L18 straight away.

Please ask your demonstrator if you are struggling with anything.

1 Dijkstra's Algorithm

Lecture 16 gave two implementations of Dijkstra's Algorithm:

- DijkstraSimple (slide 24), where we compute the optimal fringe edge using a straightforward "double loop" evaluation over all $u \in S$ and all $(u, v) \in E(G)$ where $v \notin S$. This algorithm has a running-time of $O(mn)$.
- Dijkstra (slides 28-29), which uses a (Min) Heap data structure to implement a more sophisticated version of the algorithm, with each iteration only considering the *new* fringe edges (u, x) for the most-recently added fringe vertex u. This implementation of Dijkstra is known to have running-time $O((n + m) \log(n))$.

In this section of Lab4, you are asked to implement *both* these methods. You are also asked to implement an `__init__` method which reads in file input and encodes this into an *Adjacency list* representation - using an Adjacency list is important, as this is necessary to achieve the stated running times above.

1.1 File input

We will assume that the format of input files are organised so that every line of the input includes 3 items separated by whitespace:

`u v w`

This line should be interpreted to mean that "There is an edge `(u,v)` with weight `w`". I have not said whether this is directed or undirected, and that information is not given in the input file - the directed/undirected status of an input file will be communicated by setting a `directed` parameter to `True` or `False` when calling the `init` constructor.

We will assume by default that the vertex set V will always be `0, 1, ...., n-1`, to match the default indexing of arraylists and other data structures in Python. Therefore we assume that the 1st two items `u` and `v` along a line of the input file are always of type `int`. For the `w` entry, you may chose whether you will require this to also be `int`, or whether to work with the more general `float` - it is up to you.

Your first task as part of this coursework is to complete the constructor method:

```
def __init__(self,n,directed,filename):
```

The key attribute that needs to be set-up in your implementation of the constructor is `self.adj_list`.

- For `self.adj_list`, we have already initialised this to be a dictionary with empty lists as the (initial) values. As we read the input file, and process input lines, we will want to add items `(v,w)` to the list `self.adj_list[u]` to represent that "there is an edge `(u,v)` with weight `w`".
- If the parameter `directed` is `True`, then each line of the input file requires us to add one edge to `self.adj_list` (add `(v,w)` to the list `self.adj_list[u]`).

  However, if we have an *undirected* input graph (`directed` is `False`), then we need to add an edge to *both* `self.adj_list[u]` and `self.adj_list[v]`.
- Much of the effort to achieve the implementation of the constructor is the reading of the input file. You will need to make use of `open`, `readlines` and `close` (for the file) and `split` (to split the input lines).
- The 1st and 2nd entry on each line must be input as `int`, it is up to you whether to use `int` or `float` for the 3rd item.
- The filename should be supplied in quotes, and remember that it must be resident in the director where you are running Python. Here is an example of me initialising with the file `7nodes`.

```
 Python 3.10.12 (main, Nov  6 2024, 20:22:13) [GCC 11.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import dijkstra
>>> g=dijkstra.Graph(7,True,"7nodes")
```

As an initial testing file, I have supplied a file called `7nodes` which encodes the edges of our example graph from L16. Given that we need to have nodes named `0, 1, ...,` I have needed to rename the nodes of the graph. The correspondence is as follows:

$$s \equiv 0, a \equiv 1, b \equiv 2, c \equiv 3, d \equiv 4, e \equiv 5, f \equiv 6$$

1.2 DijkstraSimple, $O(mn)$ running time implementation

Now we are ready for implement the simple, less efficient, version of Dijsktra's algorithm. DijkstraSimple is presented on slide 24 of L16. At each iteration, we compute the optimal fringe edge using a straightforward "double loop" evaluation over all $u \in S$ and all $(u, v) \in E(G)$ where $v \notin S$.

`dijkstra.py` contains a starting implementation of the function

```
def DijkstraSimple(self,s):
```

The few lines written are there to initialise the key variables and arrays that are crucial for the method (you will want to add some others). I have also included the `return` at the end to show the order in which we want to return the completed arrays (please edit out during debugging if that helps).

Note that `s` is the source node for the call.

The function should return the arraylist of shortest path distances `dists` (from `v`) as well as the arraylist of pointers `pi`.

Some useful notes:

- You will need to keep track of the nodes in S and those in $V - S$ during the execution of the algorithm. You may use the Python `set` data type for this. A `list` can also be used, but the `set` is closer to the pseudocode for the algorithm, as S and $V - S$ are sets.

Once you have a working version of `DijkstraSimple`, next step is to test on some real inputs. You may want to experiment with `7nodes` in order to compare to the solution we solved in L16. Here are the results from my own implementation.

```
[archlute]mcryan: python
Python 3.10.12 (main, Nov  6 2024, 20:22:13) [GCC 11.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import dijkstra
>>> g=dijkstra.Graph(7,True,"7nodes")
>>> g.DijkstraSimple(0)
([0, 4.0, 7.0, 4.0, 3.0, 11.0, 9.0], [None, 4, 1, 0, 0, 4, 4])
```

Two things to note about my results:

- The reason the values in `dists` are floating-point is because I took the decision to allow weights to be `float`. Entirely up to you if you prefer `int`.
- The choice of fringe edge to add b (now 2) into S was different from the choice we made in class (remember we had 2 competing options at this iteration).

1.3 Dijkstra using a Heap

The second implementation for Dijkstra's method was given in slides 28-29 of L16, and uses a (Min) Heap data structure, together with more careful consideration of fringe edges, to

achieve running-time $O((n+m)\log(n))$.

In this task, we ask you to achieve an "approximate" implementation of the Heap version Dijkstra. You are not required to explicitly code up Relax as a separate method, that is up to you (and in my own implementation I wrote all the code in the main method).

In implementing Heap Dijkstra, you need to make use of the Python `heapq` library, which is documented at the following website.

- `heapq` differs somewhat from our presentation in Lecture 11 (as we mentioned at the time, and in the related tutorial). For one thing, it is a Min Heap (which helps us, as we need Min for Dijkstra).
- The main methods you will need to exploit in implementing Dijkstra will be `heappush` (corresponding to insertItem) and `heappop` (corresponding to extractMin).

There is one problem with using `heapq`, which is that `heapq` (like most basic implementations of a Heap), does not include a reduceKey operation.

- Your "workaround" to this should be to *instead* "re-insert" the node v *again* into the Heap, with the improved key value (the better $d[v]$). You will need to make some other changes to the implementation to get everything to work "as a whole".
- It is possible to re-engineer `heapq` to include a "true to $O(\log(n))$" reducekey. However, it requires us to introduce a new index into the Heap that must be maintained by all methods, and hence requires a good amount of re-working of the `heapq` library.

  I didn't have time to do this. If anyone would like to have a go, please do! (and chat with me)

`dijkstra.py` contains a starting implementation of the function

```
def Dijkstra(self,s):
```

The few lines written are there to initialise the key variables and arrays that are crucial for the method (you will want to add some others). I have also included the `return` at the end to show the order in which we want to return the completed arrays (please edit out during debugging if that helps).

Note that `s` is the source node for the call.

Once you have a working version of `Dijkstra`, you can again experiment with `7nodes` in order to compare to the solution we solved in L16. Here are the results from my own implementation (exact same as for DijkstraSimple).

```
[archlute]mcryan: python
Python 3.10.12 (main, Nov  6 2024, 20:22:13) [GCC 11.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import dijkstra
>>> g=dijkstra.Graph(7,True,"7nodes")
>>> g.Dijkstra(0)
([0, 4.0, 7.0, 4.0, 3.0, 11.0, 9.0], [None, 4, 1, 0, 0, 4, 4])
```

2 Coin changing

In this section we ask you to code-up algorithms for the "coin changing" problem, and run some experiments comparing the run-times with different implementations.

The template file `coin_changing.py` is a very minimalist starting point, which essentially has some method declarations, and little else. The intention is that you will develop the code in an imperative way, without making use of object-oriented structure. Your methods will work with respect to a global variable `c_list` which can be set to the specific coin system you are working with (`c_list` will be a list containing the different coin values, in increasing order).

2.1 Finding the minimum number of coins (recursively)

In Lecture 18, we presented a *recurrence* for solving the coin-changing problem on slide 10. It is straightforward to use this recurrence to quickly implement a *recursive* method to compute the minimum possible number of coins for a given value v. You should implement a recursive solution to find that minimum number of coins as the following method:

```
def fewest_coins(v):
```

Of course, we really want to know the/a specific collection of coins which will realise this "minimum possible number of coins". This can also be solved recursively, with the output to this method being a list of coin values. You should implement this in the following method:

```
def fewest_coins_list(v):
```

2.2 "Memoization" to improve running-time

The concept of *memoization* has come up a couple of times during lectures, and in Live Discussions. This concept is an alternative way of avoiding repeated computation of subproblems - instead of explicitly identifying all potential subproblems that might ever arise (and building a table with all these solutions computed in a careful order), *instead* we could apply a recursive approach, but make a copy (a "memo") of the result for each subproblem every time it is computed for the first time. The recursive algorithm will then check the memo for a stored solution before it makes the recursive call(s) it would normally make.

For a single-argument function such as the two above, we can use the exact same memoization function as we saw in the lectures.

```
def memoize(f):
      memo = {}
      def check(v):
            if v not in memo:
                memo[v] = f(v)
            return memo[v]
      return check
```

`coin_changing.py` contains this function, as well as (commented-out) calls to exploit memoize with the methods you have now code-up for solving coin-changing.

Try some example calls, and use `timeit` to investigate the difference in running-times between the naïve recursion and the memoized variant.

2.3 The dynamic programming approach

We saw a dynamic programming algorithm Dyn-Coin to solve this problem at the end of lecture 18, both finding the minimum number of coins and also constructing a set of coins to achieve this. Your next task is to implement dynamic programming solutions for the methods:

```
def fewest_coins_dp(v):

def fewest_coins_list_dp(v):
```