Lab: Search

Instructions

1. Complete Tasks 1 and 2, as detailed below.
2. Re-zip the lab with your changes
3. Submit it to Moodle

Important: Do not change any existing file names or the directory structure **because this may affect the automatic component of your grade** (testing your code). Use Python 3. Re-zip the files into `lab3.zip` with no directory inside (i.e., as you found the `zip`). Your code must be documented/commented sufficiently to be understood easily.

UPDATE 24/01/2019: NOTE REGARDING TASK 1: This is designed as a search problem. In the earlier formulation, Figure 1 showed an extra two tiles which makes a search too easy (in fact, a greedy search can be optimal, simply locally-choosing each tile to fit its surroundings). In this updated version (you may *only use tiles shown* in Figure 1 of *this version* – noting that two are missing), such an approach is not necessarily possible.
In the spirit of a search task, you should have designed a search from the lecture material, as specified. However, some already proceeded with a greedy solution (thanks to those who pointed out that this was possible given the initial tileset!) If you have already completed such a solution, you may include it as a separate function `gen_map_fast(M)` which may be considered for a small extra credit. However, you must adapt your solution (in `gen_map`) to a search method, taking into account the absence of certain tiles.
Additional clarification has been added (and highlighted) in the description of this task, for example, regarding tile separation; as this was also requested.

Task 1

Many tile-based games require maps to be edited by hand in a map editor, placing tiles one at a time. You task is to generate automatically valid maps from a tileset. You are given some open-source game-art tiles as shown in Figure 1.

```
11 01 10 00 11 01 00 00 11 00 10 00 11 01 10 00
11 11 11 11 01 01 00 01 10 00 10 10 00 00 00 00
```

Figure 1: A game tileset as images (top) and corresponding tile code (below); land is specified by `1` and water by `0`. Note that three full-water tiles are repeated.

Your task is as follows: generate a $m \times n$ map which is valid in the sense that tiles are correctly joining. This means that land should have a shoreline when joining the water; and correspondingly – bits should match to those of neighbouring tiles. Consider the following examples of joining two tiles, where the left example is valid and the right example is invalid (because a `0` neighbours with a `1` on the adjoining tile below).

```
          11
1110      00
0000      01
          11
```

An example of a correct map is shown in Figure 2 (centre) which is represented as the following binary matrix:

```
1110000000
0000000000
0000000000
0001111000
0001111000
1001111000
```

Note that each tile is represented by a grid of 4 bits, where `1` indicates land, and `0` water, and `-1` where no tile has been placed yet (in the case of an incomplete map).

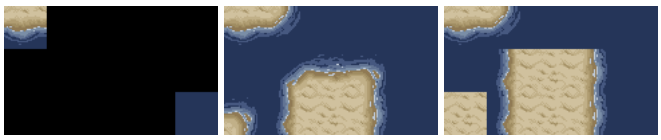

Figure 2: An example incomplete map (left), valid map (centre), and invalid map (right).

In `map_generator.py` you are given an empty function `gen_map` to complete, noting the following constraints and hints:

- Do not change the filename, name of the function or the parameters.
- You can assume that the *given* map is *incomplete* and *invalid* and may be different each time (including different size).
- You can infer the dimensions of the map via `M.shape` (divide by 2 to obtain the number of game tiles in each dimension)
- You must place tiles to complete the map without removing any tiles, i.e., replace the missing tiles denoted by a grid of `-1` only using the tiles shown in Figure 1.
- You should implement some **search** method mentioned in the lecture, but you should decide which is most suitable.

Include a `README.txt` explaining very briefly (1 paragraph max.) which algorithm you chose. Grading is based on

1. Automatic evaluation (time, correctness)
2. Choice of algorithm/approach (describe in `README.txt`)
3. Validity/quality of code

A score of 8/10 can be obtained by providing a relatively simple approach which successfully completes a relatively small map (like 6×6) in a few minutes; if code successfully runs and instructions were followed. Remaining score is for particularly efficient or elaborated search methods.

Task 2

In `run_task2.py` there is code which trains a classifier chain $\mathbf{h} = [h_1, \ldots, h_L]$ of L single-label models $h_j : \mathcal{X} \to \{0, 1\}$ (one model for each label) on a given dataset $\{\mathbf{X}, \mathbf{Y}\}$. The classifier chain method itself is implemented in `cc.py`. For each instance $\mathbf{x} = \mathbf{X}_{i,:}$ (row) of the test data, the `predict` function calls a function named `explore_paths(self,x,epsilon)`. The implementation of this function as it is given carries out a greedy search on the probability tree, returns the branches explored, the predicted path $\mathbf{y}$ corresponding to these branches to the goal node (the best path of those explored), and the payoff incurred (note: higher payoff is better) according to payoff function $P(\mathbf{y}|\mathbf{x}) = \prod_{j=1}^{L} P(y_j|\mathbf{x}, y_1, \ldots, y_{j-1})$ where each P_j corresponds to the conditional posterior of each j-th classifier in the chain (logistic regression). This is the equivalent to ϵ-approximate search with $\epsilon > 0.5$ (although, note: the `epsilon` parameter is not used *yet*), but your task is **to complete this function** so it is correct for any ϵ (given as `epsilon`).

The list of `branches` returned by this function can be used to show the search/exploration. Indeed, there is a function in `utils.py` which converts this to GRAPHVIZ markup and writes it to a file that may be later visualized graph. You may view it by installing GRAPHVIZ or using an online tool like `http://www.webgraphviz.com/` (simply copy and paste the contents of the file there). An example of the result for a particular $\mathbf{x}$ is given in Figure 3. Of course, your completed code will output the path explored by ϵ-approximate search.

The task is to re-implement the function `explore_paths` in `cc.py` as ϵ-approximate search, using the `epsilon` parameter provided. Make sure to also return the *best* path (wrt $0/1$-loss) and its score as suggested in the current implementation – such that the `predict` function works correctly.

Your grade (/10 for this task) is based on

- The predictions obtained by your implementation of `explore_paths` (for a given random `seed` and `epsilon`)
- The correctness of the resulting path (for a given instance and `epsilon`)
- Your implementation (i.e., your code)

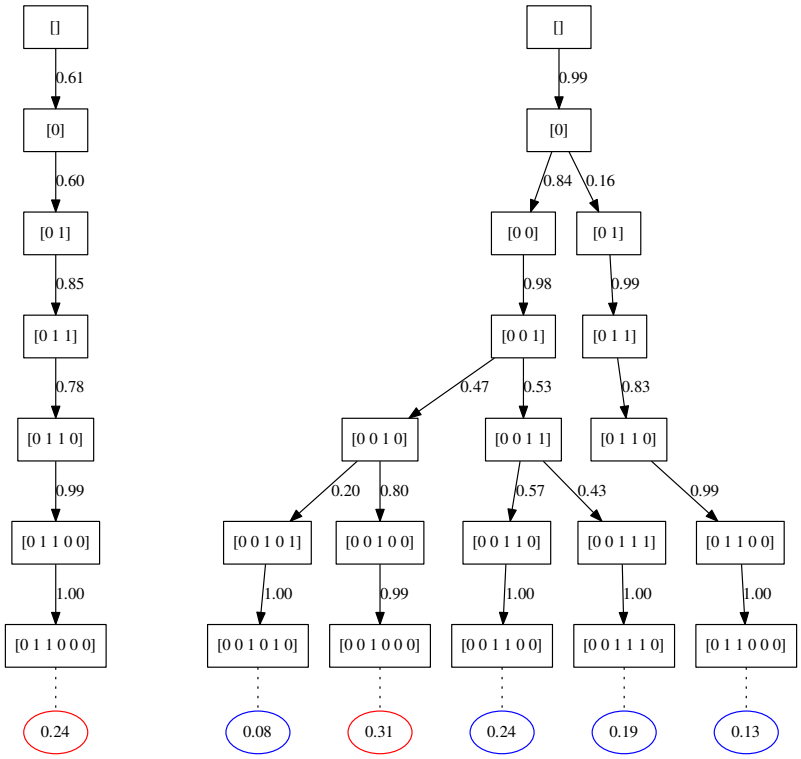

Figure 3: The paths explored by greedy search (left) and Monte Carlo sampling (right) for a given instance. Each explored node is represented by the accumulated path in a rectangle, and the value of each edge is the payoff. The red oval node attached to the goal node(s) displays the overall path cost (it is not part of the tree; unexplored paths are not shown).