TD9. Union-Find: some case studies

November 19, 2020

In this tutorial, we will study the Union-Find data structure and some of its applications. Last year, you saw applications in graphs for computing minimal spanning tree (with the Kruskal algorithm) and for maze generation. We will see how we can use union-find for the study of properties of a model of random graphs, estimate percolation probability in two dimensional grids and for checking the winning condition of the Hex game.

Open a new project `CSE202_TD9` in Spyder. The file `testing.py` contains several test functions to help you with debugging (calls to the successive testing functions are gathered at the end of that file; for each question you can uncomment the line of the corresponding call before running `testing.py`).

1 An efficient Union-Find implementation

We recall that Union-Find is a data structure used for representing equivalence classes of some data. For simplicity, we assume that data are disctinct integers from 0 to $N-1$.

Internally, the data structure uses a forest representation for the equivalence classes via an array A indexed by the data integers. Each entry of this array is one of the other data integer: if $q = A[p]$ is the entry at index p, then q is the parent of p. If $A[p] = p$, then p is the root of one of the tree of the forest. The equivalence class of p is the root of its corresponding tree, and each equivalent class is represented by one of the tree's root of the forest. Figure 1 illustrates such a representation for $N = 12$ and equivalence being based on the modulus of 3.

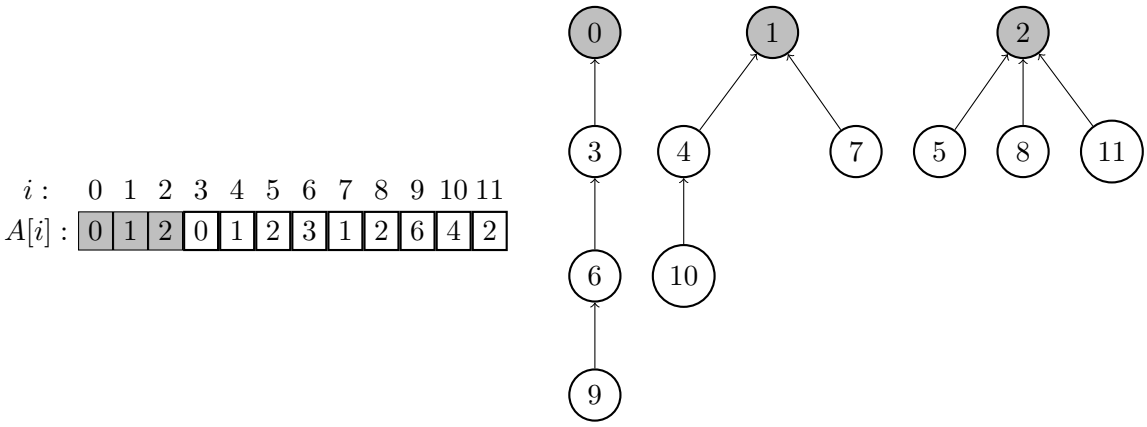

Figure 1: Array representing a forest of trees. Each tree correspond to an equivalence class (represented by the root element in gray). Here, equivalence classes are with respect to the modulus of 3.

A Union-Find data structure holding N integers is initialised to a forest where every p in $0, \ldots, N-1$ is a root. The structure then typically implements the following operations

- A function `union(p,q)` which merges the equivalence classes of p and q into a single one: the root of p becomes the root of the root of q, or vice-versa.
- A function `find(p)` which finds the element representing the equivalence class of p, *i.e.* finding the root of the tree where p belongs.
- Optionally (but handy), a function `is_connected(p,q)` which checks whether p and q belong to the same equivalence class, *i.e.* if their corresponding respective roots are the same.

The file `uf.py` contains a simple implementation of Union-Find with the class called `UF_base`. It contains three fields: `N` (the number of elements), `A` (the array representing the forest) and `count` which is an integer corresponding to the number of equivalence classes (number of trees in the forest). The Union-Find operators are defined following the *quick-find* implementation : the `find` operation has $O(1)$ time complexity, however `union` has $O(N)$ time complexity.

You will have to implement the sub-class `Rank_UF`, a more efficient version of Union-Find: *union by rank with path compression* (the version of the course). It has an extra field `rank` which is an array indexed by i in $0, \ldots, N$: `rank[p]` contains the height of the sub-tree rooted at p.

Question 1 Implement the function `union(p,q)` of `Rank_UF` for the union with rank: the class absorbing the other is the one with highest rank (the rank increases by one if necessary).

Question 2 Implement the function `find(p)` of `Rank_UF` for find with path compression.
Note. You can test your implementation with the function `test_random_uf` and `test_rank_uf` from `testing.py`.

2 Case studies

2.1 Analyzing the Erdős-Renyi model of random graph

Models of random graphs are useful to test the performances and robustness of graph algorithms. Erdős and Renyi have proposed a simple model for random graphs: $G(N, p)$ designates the set of graphs with N vertices where each possible edge (with distinct endpoints) have a probability p to appear in the graph. As there are a maximum of $\binom{N}{2}$ possible edges, a graph from $G(N, p)$ have approximately $\binom{N}{2}p$ edges.

A question we can ask is for an arbitrary large N, which value of p make most of the graphs from $G(N, p)$ connected (there exist a path between any pair of nodes). The theory says that $p^* = \dfrac{\log(N)}{N}$ is a threshold for the connectedness of the graphs: *e.g.* if $p > p^*$, graphs in $G(N, p)$ are likely to be connected.

We will estimate experimentally this threshold using Union-Find. We will consider simulating the random generation of a graph following the model of Erdős-Renyi. A Union-Find data structure will be used for storing the connected components of the generated graph. The file `erdos.py` contains the signature of the function you need to implement with an additional test function to display your result.

Question 3 Implement the function `Erdos_Renyi(N)` which simulates the construction of a random graph, stopping once it is connected. It should implement the following steps:

1. Initialize a Union-Find data structure of size N associating to the nodes of the graph.
2. While there are more than one connected component:

i. generate a random pair of distinct nodes (p, q).

ii. if they are not connected, do the `union` of these nodes.

3. Return the number of generated pairs of nodes.

We should expect to have generated about $\binom{N}{2}p^* \approx \frac{N\log(N)}{2}$ edges before termination. Check this by calling the function `test_erdos()`.

2.2 Percolation on a square grid

We consider a 2-dimensional $N \times N$ grid of *vacant* or *non-vacant* cells. We say there is *percolation* if the top of the grid can attain the bottom the grid through a path of adjacent vacant cells. Figure 2 illustrates one case of non-percolation and one case of percolation on a 6×6 grid.

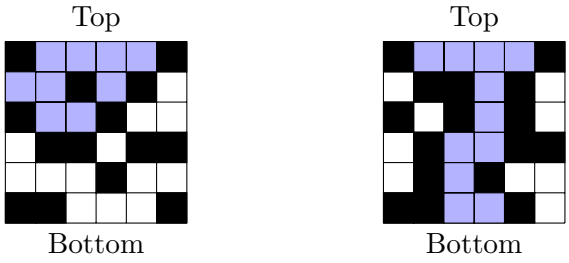

Figure 2: Black cells are non-vacant, white are vacant and blue are vacant and connected to the top. (Left) No percolation from the Top to the Bottom (Right) Percolation.

We denote by p the probability that an individual cell of the grid to be vacant. We would like to estimate the probability threshold p^* of vacancy above which percolation of the grid of size $N \times N$, with N arbitrarily large, is highly likely. This problem relates to *percolation theory*[1] which has many applications in mathematics and physics for the study of materials. In the context of the percolation on the 2-dimensional grid, there is currently no known mathematical solution for finding p^* but its value can be experimentally estimated. We will do it using Union-Find.

Download the file `perco.py` which contains the signature of the functions you have to implement, plus some extra functions to help you. We will represent the grid by a double array with $N + 2$ lines and N columns. Each cells of this double array contain either `False` if it is non-vacant and `True` otherwise. The first and last row corresponds respectively to the top and bottom of the grid, and should be all vacant.

Note. Use the function `pos_to_int(N,i,j)` to associate a position of the grid to an integer for the Union-Find.

Question 4 Implement the function `get_vacant_neighbors(G,N,i,j)` which given a grid G, its size N and a position (i, j) returns the list of positions of all adjacent cells which are vacant.

Question 5 Implement the function `make_vacant(UF, G, N, i, j)` which given a union-find object `UF`, a grid G (of size N) and a position (i, j), sets vacant the cell at (i, j) and do its union with all vacant neighboring cells within `UF`.

Note. You can test your implementation with the function `test_get_neighbors` and `test_make_vacant` from `testing.py`.

[1] see `https://en.wikipedia.org/wiki/Percolation_theory`

Question 6 Write the function `ratio_to_percolate(N)` which given a integer N:

1. generates an $(N+2) \times N$ grid of non-vacant cells (first and last row all vacant).
2. initialise an Union-Find object of size $(N+2)N$ and do the union of the first and last row of the grid (independently)
3. While the top and bottom are not connected (check using the Union-Find object):
   i. randomly select a position (i, j) within the grid (which does not belong to the first and last row)
   ii. if the corresponding cell is not vacant, make it vacant using `make_vacant`
4. Returns the ratio of vacant cells (not counting first and last row).

Run your algorithm multiple times for various value of N. What is your estimate for p^* ?

2.3 The Hex game

Hex[2] is a board game where two players (red and blue) play on a hexagonal grid (usually of size 11×11). Each player possesses one side and its opposite side of the board. At his turn, a player put a token of its color on one of the free hexagons of the board. A player wins the game by building a path of its color joining each of its side of the board. Note that the game can never end in a draw. We will use Union-Find to monitor the plays of the game and detects when a player has won. Figure 3 depicts an example of game played on a 6×6 hexagonal grid.

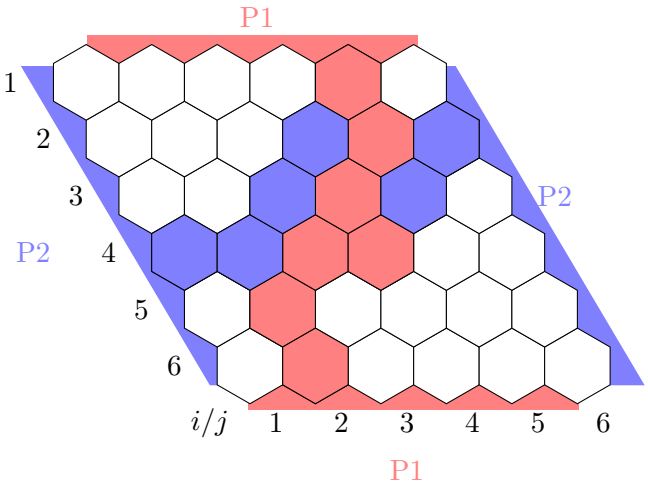

Figure 3: Hex game. Configuration where the player 1 (in red) wins.

Download the file `Hex.py` which contains the structure of a class `Hex` for the Hex game. The board is represented by the attribute `board` which is a double array with $N+2$ rows and $N+2$ columns where each entry is either 0 (meaning the hexagon is free), or 1 or 2 if respectively player 1 or 2 has put a token there. There are two extra rows and two extra columns representing each opposing side of each player (hence, they are initially considered full with their respective tokens). Player 1 possesses the two extra rows and player 2 the two extra columns.

The other attributes of `Hex` are `uf` which is the union-find object that will monitor the game, `size` which is the length of each side of the board (equal to $N+2$), `player` being 1 or 2

[2] `https://en.wikipedia.org/wiki/Hex_(board_game)`

depending on which player is playing the current turn and `boti`,`topi` ($i = 1, 2$) being the index (for the Union-Find) of each opposing sides of the board belonging to player 1 or 2.
Note. Use the function `hex_to_int(N,i,j)` to associate a position of an hexagon of the board to an integer for the Union-Find.

Question 7 Implement the function `neighbours(self, i,j)` of the class `Hex` which computes and returns the list of the hexagons that are neighbors of the hexagon at position (i, j) and belong to the current player.

Question 8 Implement the function `is_game_over(self)` which returns `True` if the current player has won the game.
Note. You can test your implementation with the function `test_hex_neighbors` and `test_hex_winning` from `testing.py`.

In order to test a game of Hex, we will simulate a simple random game.

Question 9 Implement the function `random_turn(self)` which performs a turn of the current player: select randomly a free hexagon and assign it to the current player. Then make the union of the selected hexagon to the neighboring ones belonging to the current player.

Question 10 Implement the function `random_game(self)` which simulates a game with random plays (using `random_turn`). The game stops once one of the player has fulfilled the winning conditions. The function should return the ratio of the number of non-free hexagons (not counting the extra rows and columns).

Observe how the much the board is filled with tokens by simulating random games for various board size.

Note. You can check your algorithm is correct for small board sizes using the function `draw_board` of the class `Hex`.