École Polytechnique — CSC_2F002_EP Design and Analysis of Algorithms

Union–Find: some case studies

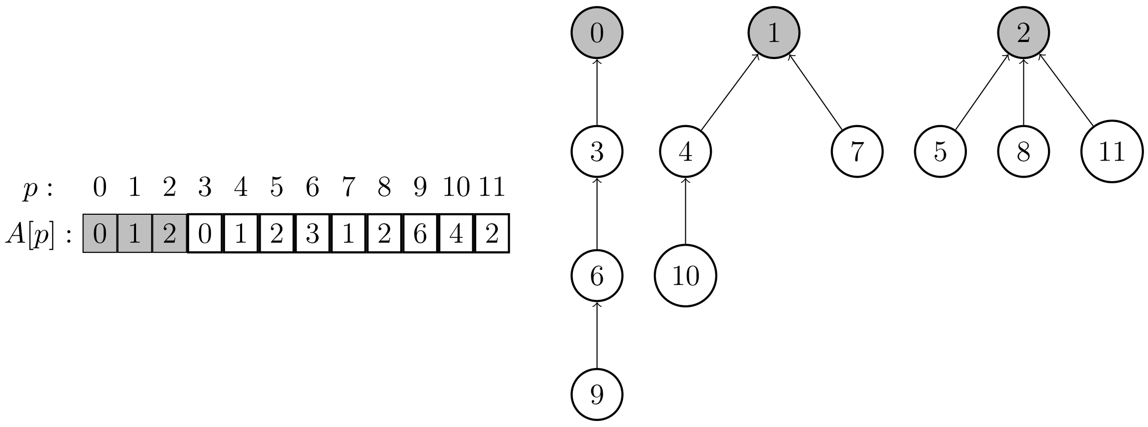
In this tutorial, we will study the union–find data structure and some of its applications. Last year, you saw applications in graphs for computing minimal spanning tree (with the Kruskal algorithm) and for maze generation. We will see how we can use union–find for the study of properties of a model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

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 distinct 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 trees in the forest. The equivalence class of p is the root of its corresponding tree. Each equivalence class is represented by the root of one of the trees in the forest. Figure 1 illustrates such a representation for $N = 12$ and the equivalence defined by the equality modulo 3.



p : 0 1 2 3 4 5 6 7 8 9 10 11
$A[p]$: 0 1 2 0 1 2 3 1 2 6 4 2
0
1
2
3
4
7
5
8
11
6
10
9

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

A union–find data structure holding N integers is initialized 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)` that 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)` that 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)` that checks whether p and q belong to the same equivalence class, *i.e.* if their corresponding respective roots are the same.
- Optionally (but handy), a function `get_count()` that returns the number of equivalence classes.

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 p in $[0 : N]$. The cell rank[p] contains an upper bound for the height of the sub-tree rooted at p (it is precisely the height when path compression is not used).

Question 1. Implement the function find(p) of Rank_UF for find **with path compression**.

You can test your implementation with the function test_find(N) from testing.py. This function takes as an argument the number N of integers in the data structure.

Question 2. 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). Make sure to not forget to update the variable count holding the number of equivalence classes.

You can test your implementation with the function test_random_union and test_rank_uf from testing.py.

2 Case studies

2.1 Analyzing the Erdős–Rényi model of random graph

Models of random graphs are useful to test the performances and robustness of graph algorithms. Erdős and Rényi 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) has 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 arbitrarily large N, which value of p makes most of the graphs from $G(N, p)$ connected (there exists a path between any pair of nodes). The theory says that $p^* = \frac{\log(N)}{N}$ is a threshold for the connectivity 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–Rényi. 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. An additional test function test_erdos(R) to display your result can be found in testing.py.

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 is more than one connected component (use get_count in Rank_UF):
   (i) generate a random pair of distinct nodes (p, q).
   (ii) if these nodes are not in the same connected component, union them.
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(R). For each N, this function averages over R repetitions.

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 and only if the top of the grid can reach the bottom of 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.

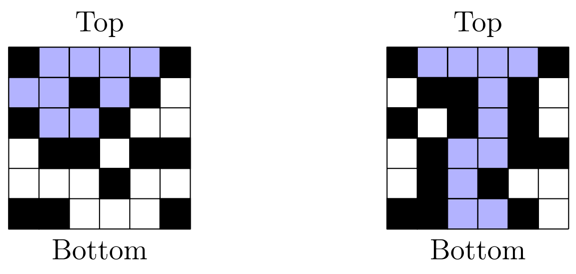


Figure 2: Black cells are non-vacant, white are vacant and blue are vacant and connected to the top. (Left) No percolation from the Top to the Bottom (Right) Percolation.

We denote by p the probability that an individual cell of the grid is 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 so 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 $N+2$ rows and N columns. Each cell of this double array contains either `False` if it is non-vacant and `True` otherwise. The first and last row, respectively, corresponds 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 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. You can test your implementation with the function `test_get_neighbors` from `testing.py`.

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 the cell at (i, j) vacant and performs its union with all vacant neighboring cells within `UF`. You can test your implementation with the function `test_make_vacant` from `testing.py`.

Question 6. Write the function `ratio_to_percolate(N)`, which, given a integer N, does the following:

1. Generate an $(N + 2) \times N$ grid of non-vacant cells (first and last row all vacant).
2. Initialize a union–find object of size $(N+2)N$ and perform the union of the cells on the first and the last row of the grid, respectively (we unite cells that are on the same row).
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^* ?

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

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 their turn, a player puts a token of their color on one of the free hexagons of the board. A player wins the game by building a path of their color joining their sides of the board. The game can never end in a draw, this might be easy to believe, but it is hard to prove.[3] We will use union–find to monitor the plays of the game and to detect when a player has won. Figure 3 depicts an example of game played on a 6×6 hexagonal grid.

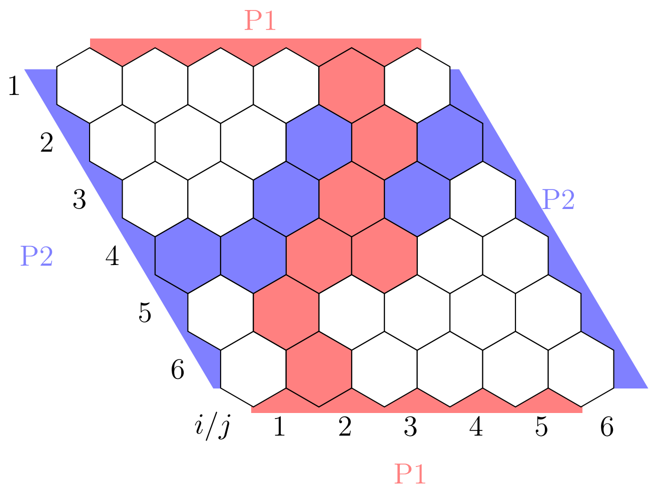


Figure 3: Hex game. Player 1 uses red tokens and possesses the top/bottom sides, and Player 2 uses blue tokens and possesses the left/right sides. Configuration where the Player 1 wins, since the top side is connected to the bottom side by a path of red tokens.

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 depending on which player is playing the current turn, and `bot[i]`, `top[i]` ($i = 1, 2$) 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 a 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. You can test your implementation with the function `test_hex_neighbors` from `testing.py`.

Question 8. Implement the function `is_game_over(self)` which returns `True` if the current player has won the game. If no player has won yet, return `False`. You can test your implementation with the function `test_hex_winning` from `testing.py`.

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

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

[3] For a “simple” proof, see Gale, David. "The game of Hex and the Brouwer fixed-point theorem." *The American mathematical monthly* 86.10 (1979): 818-827. `https://www.jstor.org/stable/2320146`

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 players has won. The function should return the ratio of the number of non-free hexagons (not counting the extra rows and columns). Observe how much of the board is filled with tokens by simulating random games for various board size. You can check that your algorithm is correct for small board sizes using the function `print_board` of the class `Hex`.