École Polytechnique — CSE202 Design and Analysis of Algorithms

Divide-and-conquer and randomization

1 Tiling by L-shapes

We consider the problem of tiling a *punctured* grid consisting of $2^n \times 2^n$ squares via what we call *L-shapes*. *Punctured* means that one square of the grid, called the *hole*, is missing. An L-shape is any set of 3 unit squares that are inside a (necessarily unique) 2×2 square. The integer n is called the *size* of the grid. The file `L_tiling.py` on Moodle contains the skeleton code that you need to complete, and the file `test_L_tiling.py` contains the code to test your solutions.

Each square of a grid is identified by consecutive cartesian coordinates, with x-coordinates increasing from left to right and y-coordinates increasing from bottom to top. The left image in Figure 1 shows an example of a punctured grid of size 2. We call the bottom left corner of a grid its *origin*, which does not necessarily need to be $(0, 0)$. In fact, it is convenient to allow the origin to be any point with non-negative coordinates. That is, for two integer $i, j \geq 0$, the x-coordinates of the grid at origin (i, j) of size n range from i to $i + 2^n - 1$, and the y-coordinates range from j to $j + 2^n - 1$.

Given integers n, i, j, a, b, we define the *punctured grid of type* (n, i, j, a, b) to be the $2^n \times 2^n$ grid with origin (i, j) and with the hole at (a, b).

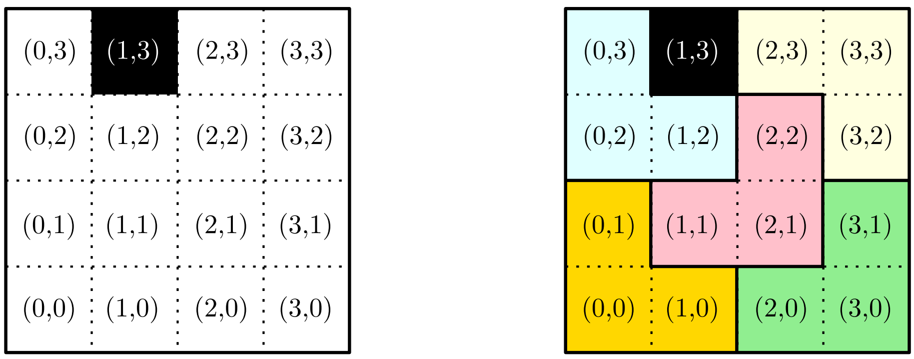

Figure 1: Left: the punctured 4×4 grid G with the hole at position $(1, 3)$. Right: an L-tiling of G.

We are interested in tiling a punctured grid, that is, cover it with non-overlapping L-shapes. The right image of Figure 1 shows an example of such a tiling. In Python, we represent a tiling as a list of triples (the L-shapes) of integer pairs. In Figure 1, the list is (up to reordering) `[[(1, 1), (2, 2), (2, 1)], [(0, 0), (0, 1), (1, 0)], [(3, 1), (3, 0), (2, 0)], [(0, 2), (0, 3), (1, 2)], [(3, 2), (2, 3), (3, 3)]]`.

In order to compute a valid L-tiling of a punctured grid G, we use the following divide-and-conquer strategy, also sketched in Figure 2. If the size n of G is zero, return. If n is greater than zero, we decompose G into 4 quadrants (each a grid of size $n - 1$). We call the *marked quadrant* the one containing the hole of G. Let the *middle* M of G be the L-shape obtained from the central 2×2 square X by removing the unique square of X that belongs to the marked quadrant (left and central image of Figure 2). We add M to our current tiling. Then we puncture the marked quadrant to have the same hole as G, whereas the other 3 quadrants are punctured at their unique unit square belonging to X (right image of Figure 2). Last, we apply the tiling procedure to each of the 4 quadrants recursively.

Question 1. Complete the function `middleL(n, i, j, a, b)`, which, given a punctured grid of type (n, i, j, a, b) with size $n \geq 1$, returns the middle of the grid as a list `[(x1, y1), (x2, y2), (x3, y3)]`. (Any order of the list is fine.)

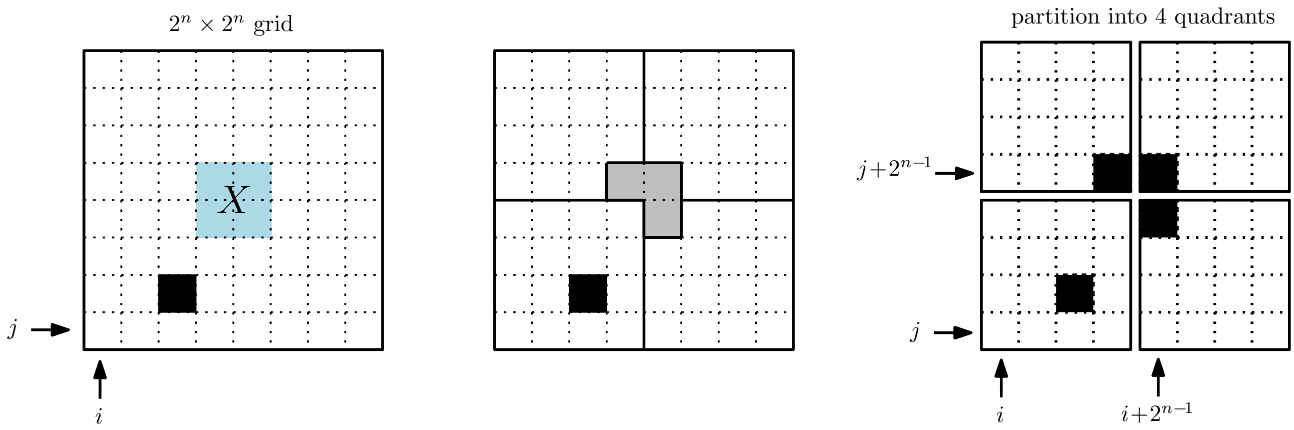

Figure 2: The divide-and-conquer strategy to compute an L-tiling of a punctured $2^n \times 2^n$ grid.

Question 2. Complete the function `lower_left_hole(n, i, j, a, b)`, which returns the coordinates `k, l` of the hole in the lower left quadrant of the punctured grid of type (n, i, j, a, b). Complete analogously the functions for the other three quadrants.

Question 3. Complete the recursive function `tile(n, i, j, a, b)`, which contains a list L-shapes that form a valid tiling of the punctured grid of type (n, i, j, a, b).

Once this function passes the test, you can uncomment the function `display_tiling_with_random_hole(n)` in `test_L_tiling.py`, which displays the L-tiling of the $2^n \times 2^n$ grid punctured at a square chosen uniformly at random.

2 Randomized MinCut

We now implement a randomized algorithm to compute the minimum cut (mincut) of a multigraph. A multigraph G is a graph where there can be several edges between any pair of vertices. There are no self-loops in a multigraph, that is, edge connects two *distinct* vertices. Figure 3 shows an example (for instance vertices c and g are connected by two edges). The *degree* of a vertex is the number of its incident edges, taking the multiplicity of edges into account. For instance, in Figure 3, the vertex c has degree 4 (and has 3 neighbors). Further, the graph has a a mincut of size is 3, obtained by taking the partition $V = \{a, c, d, g\} \cup \{b, e, f, h\}$.

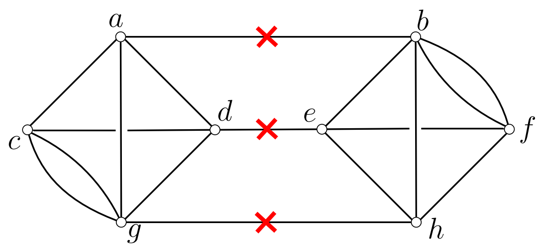

Figure 3: A multigraph and its unique mincut.

The file `MultiGraph.py` on Moodle contains the skeleton code that you need to complete, and the file `test_mincut.py` contains the code to test your solutions. The file `MultiGraph.py` contains the class `MultiGraph`, whose instances represent a multigraph. Each instance `M` has an attribute `adj`, which is a dictionary that maps each vertex v its neighbors, and each such neighbor w is mapped to the number of edges between v and w. More precisely:

- Testing if `x` is a vertex of `M` is done by `if x in M.adj`. If the test occurs in a method of the class `MultiGraph`, then this test may need to be written as `if x in self.adj`.

- If `x` and `y` are two vertices, testing if `x` and `y` are connected by at least one edge is done by `if y in M.adj[x]` If true, `M.adj[x][y]` returns the number of edges connecting `x` and `y`.

- Enumerating the set of pairs (neighbor, number of edges to that neighbor) for a vertex `x` can be done with `for (y,n) in M.adj[x].items()`.

Further, there is an attribute `deg` such that if `x` is a vertex of `M`, then `deg[x]` returns the degree of `x`. The constructor of `MultiGraph` receives a list, such as `L = [3, [['a', 'b', 1], ['c', 'b', 2], ['a', 'c', 4]]]`, where `L[0]` returns the number of vertices of the multigraph to create, and `L[1]` returns the list of edges, including multiplicities (in this example, there are 4 edges connecting `a` and `c`). An example of such a list is given by `L_tutorial` at the beginning of `test_mincut.py`, which represents the multigraph of Figure 3. Last, each instance of `MultiGraph` has a method `display`, which outputs the graph in text form to the console.

For the convenience of testing, the function `two_clique_graph(n)` in `test_mincut.py` returns (written in this list format) a graph on $2n$ vertices, for which we know the unique mincut. This graph consists of two connected cliques (that is graphs that are fully connected). More precisely, there is a clique for vertices with labels in $[1..n]$, and there is another clique for vertices with labels in $[n+1..2n]$. In addition, there are $n-2$ edges, which, for i from 1 to $n-2$, connect vertex i to vertex $2n-i+1$ (see Figure 4 for an illustration). This graph has mincut size $n-2$, which is uniquely obtained for the partition $[1..n] \cup [n+1..2n]$.

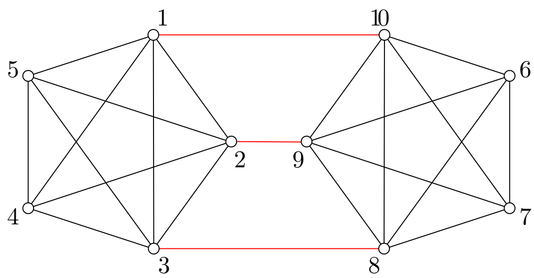

Figure 4: The two-clique graph for $n = 5$.

We now focus on the contraction algorithm seen in class. Recall that the operation of *contracting an edge* $\{u, v\}$ in a multigraph M is defined as follows: The *contracted multigraph* $M/\{u, v\}$ is obtained from M by merging u with v, keeping u as the name of the merged vertex, deleting all the edges connecting u to v (that have become self-loops), and redirecting all edges from v to u. We say that u has *absorbed* v. See Figure 5 for examples.

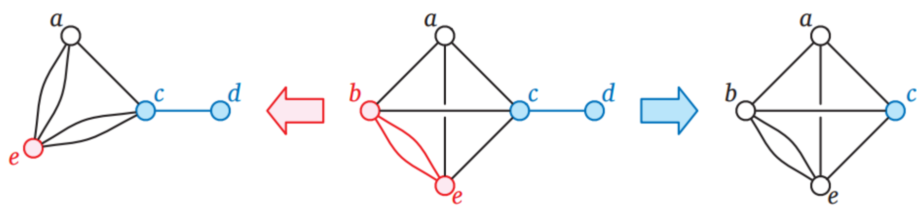

Figure 5: A multigraph M and the two contracted multigraphs $M/\{e, b\}$ and $M/\{c, d\}$.

Question 4. Complete the method `contract(self, i, j)` in the class `MultiGraph`, which contracts an edge $\{i, j\}$. Do not forget of delete the self-loops arising from the contraction and to update the attributes `deg` and `adj`. Deleting edges between two vertices `i` and `j` of a multigraph `M` can be done with `del M.adj[i][j]` and `del M.adj[j][i]`. Using `del`, it is also possible to delete the entire entry of a node in the dictionary `adj` or `deg`.

A second ingredient in the randomized mincut algorithm is to be able to draw an edge uniformly at random in a multigraph. To this end, we write a general-purpose function that draws a random element in a set of weighted elements.

Question 5. Complete the method `random_element(dict)`, which is given a dictionary `dict` whose values are positive integers, denoting the weights of each element, and returns a random element with probability proportional to the weight of the element. For instance, for `dict = {'a': 2, 'b': 1, 'c': 4}`, a call to `random_element(dict)` should return `'a'` with probability 2/7, return `'b'` with probability 1/7, and return `'c'` with probability 4/7.

Question 6. Complete the method `random_vertex(self)`, which returns a random vertex of `self`, where the weight of each vertex is its degree. For instance, in Figure 3, the sum of all vertex degrees is 34 (twice the number of edges), and the vertex `'b'` has degree 5. Hence, a call to `self.random_vertex()` should return `'b'` with probability 5/34.

Further, complete the method `random_edge(self)`, which returns an edge `(i, j)` taken uniformly at random (think of a random edge (i, j) as obtained from a random vertex i and then a random neighbor j of i). For instance, in the graph of Figure 3, there are 17 edges in total, two edges between `'c'` and `'g'`, and one edge between `'a'` and `'d'`. Hence, for a call to `self.random_edge()`, the probability that the output is `('c', 'g')` or `('g', 'c')` should be 2/17, and the probability that the output is `('a', 'd')` or `('d', 'a')` should be 1/17.

For a multigraph M with n vertices, a random cut of M is obtained as follows: For i from 1 to $n-2$, we choose an edge uniformly at random in the current multigraph (which has $n-i+1$ vertices) and contract it. At the end, there remain two vertices a, b and a bunch of $k \geq 1$ edges connecting them. This naturally corresponds to a partition $(S, \bar{S})$ of cutsize k (note that S consists of a and all the vertices that have been absorbed by a).

Question 7. Complete the method `random_cut(m)`, which returns a random cut of the multigraph `m` after contracting $n-2$ random edges as described above. The format of the output should be `[c, L]`, where `c` is the cutsize and `L` is the list of vertices in one of the two parts of the partition $(S, \bar{S})$ of the cut. In order to return `L`, you should maintain at each step a dictionary `partition` such that for each `x` not already absorbed, `partition[x]` is the list of vertices that have been already absorbed by `x`. At the end, `partition` has just two keys, `a`, `b`, corresponding to the two remaining vertices. Thus, and one can return `L` as `[a] + partition[a]`).

The probability that `random_cut(m)` returns a mincut of a multigraph `m` with n vertices is at least $\frac{2}{n(n-1)}$. As seen in class, we can boost the success probability by repeating the process k times, and output a cut of smallest size among the k experiments. Then, the probability of returning a mincut is at least $1 - (1 - \frac{2}{n(n-1)})^k$. Hence for any fixed $p \in (0, 1)$, this probability is guaranteed to be at least $1 - p$ for $k = \lceil \ln(p)/\ln(1 - 2/(n(n-1))) \rceil$ (which is $\mathrm{O}(n^2 \log(1/p))$).

Question 8. Complete the function `mincut_karger(L, p)`, which receives as input a multigraph `L` (in the list format) and a floating-point number `p` $\in (0, 1)$ and returns a cut of the multigraph such that the returned cut is a mincut with probability at least $1 - p$. In order to test your code, you can run `test_mincut_karger()`.

Fun fact. You can run the method `test_mincut_brute()` if you want to see how slowly a brute-force solution is. This method takes an optional parameter, which determines the value n of the graph from Figure 4. By default, the method chooses $n = 9$.