CSC-1S003-EP Introduction to Algorithms

TD 5: Graph Algorithms

March 13th, 2025

Exercise 1: Decoding encodings

Recall the two ways of encoding graphs that we saw in class:

adjacency list: a list of lists `G` such that `G[i]` is the list of all nodes `j` such that there is an edge `i` $\to$ `j`;

adjacency matrix: a list of lists `G` such that `G[i][j]` is `1` if there is an edge `i` $\to$ `j`, and `0` otherwise.

In both cases, we are assuming that the nodes of the graph are consecutive integers, starting at `0`. For example, for the following graph

$$\begin{tikzcd}
0 \arrow[r] \arrow[d] & 1 \arrow[loop right] & 2 \\
3 \arrow[ur] \arrow[r, bend left] & 4 \arrow[l, bend left] &
\end{tikzcd}$$

- the adjacency list is `[[1,3],[1],[],[1,4],[3]]`
- the adjacency matrix is

```
[[0,1,0,1,0],
 [0,1,0,0,0],
 [0,0,0,0,0],
 [0,1,0,0,1],
 [0,0,0,1,0]]
```

Write two Python functions `adjLstToMat` and `adjMatToLst` which convert between the two representations. For instance, if `L` and `G` are the adjacency list and adjacency matrix of the above graph, respectively, then `adjLstToMat(L)` must return `G` and `adjMatToLst(G)` must return `L`.

Exercise 2: The list of weighted edges

As we saw in class, a graph whose edges are weighted in $\mathbb{N}$ (non-negative integers) may be represented as an adjacency matrix `G` with elements in $\mathbb{N} \cup \{\infty\}$: the entry `G[i][j]` is the weight of the edge `i` $\to$ `j`, or ∞ if such an edge does not exist. Since we don't have ∞ in Python, we may use the value `-1` in our implementations.

Write a Python function `getEdges` which, given in input the adjacency matrix `G` of a weighted graph, returns the list of all triples `(w,i,j)` such that `G[i][j] == w` and `w != -1`. For example, if `G` is the adjacency matrix of the following weighted graph

$$\begin{tikzcd}
0 \arrow[r, "9"] \arrow[d, "7"'] \arrow[dr, bend left, "5"] & 1 \arrow[d, "2"] \\
2 \arrow[r, "1"'] & 3 \arrow[ul, bend left, "3"]
\end{tikzcd}$$

then `getEdges(G)` must return the following list (or a permutation of it):

```
[(9, 0, 1), (7, 0, 2), (5, 0, 3), (2, 1, 3), (1, 2, 3), (3, 3, 0)]
```

Exercise 3: Stacks and queues

Stacks and queues are two commonly used data structures. They both consist of an ordered collection of elements to which one may add an element `x` via a `push(x)` operation and extract an element via a `pop()` operation, but they differ in how such operations behave:

- popping an element from a non-empty stack will return the *last* element pushed into it;
- popping an element from a non-empty queue will return the *first* element pushed into it.

We say that stacks follow a LIFO discipline (last-in-first-out), whereas queues follow a FIFO discipline (first-in-first-out). Notice that stacks and queues both differ from arrays (or lists) because the only way to access their elements is via `pop()`, which returns the last or first element added and removes it from the structure. It is impossible to directly access, say, the third element added without having removed the two elements that were added after or before it. By contrast, arrays offer so-called random access: by writing `l[i]` one may access an arbitrary element of the array `l`, without removing it from `l`.

The `append` and `pop` methods of Python lists offer a simple way of implementing both stacks and queues. Here is a possible implementation using classes:

```python
class stack:
    def __init__(self):
        self.l = []
    def isEmpty(self):
        return self.l == []
    def push(self, x):
        self.l.append(x)
    def pop(self):
        return self.l.pop()

class queue:
    def __init__(self):
        self.l = []
    def isEmpty(self):
        return self.l == []
    def push(self, x):
        self.l.append(x)
    def pop(self):
        return self.l.pop(0)
```

One may verify that executing the code

```python
s = stack()
for i in range(5):
    s.push(i)
for i in range(5):
    print(s.pop(), end=" ")
```

prints

```
4 3 2 1 0
```

whereas executing the code

```python
q = queue()
for i in range(5):
    q.push(i)
for i in range(5):
    print(q.pop(), end=" ")
```

prints

```
0 1 2 3 4
```

However, while the above implementation of stacks is efficient (both push and pop are $\Theta(1)$ operations), the same does not hold for queues, because a call to l.pop(0) in Python costs $\Theta(n)$, where n is the length of l.

A simple workaround is to keep a pointer h to the head of the queue and, when pop() is invoked, we return l[h] but, instead of removing it, we simply increase h. This provides a constant-time pop() operation, with the downside of making the underlying list grow arbitrarily big.

Write a Python class implementing queues as suggested above, also adding a compress() method which, when called, removes the elements at the beginning of the list which are not used anymore (that is, those before h), in order to save space when needed. The compress method must work *in place*: executing compress() must not change the location where the list l underlying the queue is stored, it must simply reduce its length (if h != 0).

To help you, here is the structure of the code you'll have to complete:

```python
class queue:
    def __init__(self):
        self.l = []
        self.h = 0
    def isEmpty(self):
        # your code here
        # must have constant complexity!
    def push(self, x):
        self.l.append(x)
    def pop(self):
        # your code here
        # must have constant complexity!
    def compress(self):
        # your code here
        # must be in place and have linear complexity!
```

To test your implementation, you may for example run the following code:

```python
q = queue()
for i in range(6):
    q.push(i)
for i in range(5):
    print(q.pop(), end=" ")
print()
print(q.l, q.h)
l = q.l
q.compress()
print(l, q.h)
```

It should print

```
0 1 2 3 4
[0, 1, 2, 3, 4, 5] 5
[5] 0
```

The first line reflects the FIFO behavior of queues. The second line shows that, although the queue now contains only one element (all the other elements have been popped, and in fact q.h points to the last element of the list q.l) the list underlying the queue still contains all the elements. The third line shows that compress has worked properly: the underlying list is now a singleton, the pointer q.h returned to the initial value, and the whole operation was done in place (this is why we print l instead of q.l: if compress had changed the location of q.l, l would still point to the original location and printing it would reveal the problem).

Exercise 4: Traversing a graph

Graph traversal (or *graph search*) is the exploration of the nodes of a graph following the structure given by the edges. There are two main modes in which a graph may be traversed:

depth first: we follow the first outgoing edge of the starting node, which leads us to a node i_1, then we follow the first outgoing edge of i_1, which leads to a node i_2, and so on, until we reach a node i_n having no outgoing edges or that we have already visited; at that point, we start the same process with the second edge of i_{n-1}, and so on.

breadth first: we first visit the nodes $i_1, \ldots, i_m$ immediately reachable from the starting node, and then we repeat the procedure with i_1 through i_m, each time visiting first the nodes immediately reachable from those nodes before moving the next depth (we must of course keep track of the nodes that have already been visited).

For example, in the tree below

$$\begin{tikzcd}
 & & & 0 \arrow[dll] \arrow[drr] & & & \\
 & 1 \arrow[dl] \arrow[dr] & & & & 2 \arrow[dl] \arrow[dr] & \\
3 & & 4 & & 5 & & 6
\end{tikzcd}$$

supposing that outgoing edges are ordered from left to right and that we start from node 0, a depth-first traversal will visit the nodes in the order $0, 1, 3, 4, 2, 5, 6$, whereas a breadth-first traversal will visit the nodes in the order $0, 1, 2, 3, 4, 5, 6$.

Write a Python function `traverse` which, given in input an adjacency matrix `G`, a node `start` and a boolean `depthFirst`, returns a list containing the nodes of the graph in the order in which they are visited during a depth-first traversal if `depthFirst` is `True`, or a breadth-first traversal if `depthFirst` is `False`, starting from `start` in both cases. During the traversal, we will assume that an edge $i \to j$ is considered before $i \to j'$ when $j > j'$ in the depth-first case, or when $j < j'$ in the breadth-first case. For example, if `G` is the adjacency matrix of the graph

$$\begin{tikzcd}
0 \arrow[r] \arrow[d] & 1 \arrow[r] & 2 \arrow[r] \arrow[d] \arrow[ll, bend right] & 3 \\
4 \arrow[r] & 5 \arrow[ul] \arrow[r, bend left] & 6 \arrow[l, bend left] &
\end{tikzcd}$$

then:

- `traverse(G, 0, True)` must return `[0, 4, 5, 6, 1, 2, 3]`;
- `traverse(G, 0, False)` must return `[0, 1, 4, 2, 5, 3, 6]`.

Observation: you may of course solve this exercise by writing something like

```
def traverse(G, start, depthFirst):
    if depthFirst:
        # implementation of depth-first traversal
    else:
        # implementation of breadth-first traversal
```

However, if you think about it, you'll see that the same *code works for both depth-first and breadth-first traversal, you just need to use a parametric type which you will instantiate as* `stack` *or* `queue`*, depending on the case. By "parametric type" we mean something like the following example:*

```
def count(n, fwd):
    if fwd:
        listType = queue
    else:
        listType = stack
    l = listType()
    for i in range(n):
        l.push(i)
    for i in range(n):
        print(l.pop(), end=" ")
    print()
```

The variable `listType` *is a parametric type, in the sense that it may be instantiated to* `queue` *or* `stack` *depending on the value of the parameter* `fwd`*. Since both* `queue` *and* `stack` *have the methods* `push` *and* `pop`*, the above code makes sense without knowing the value of* `listType` *in advance. You may try it and see that* `count(5, True)` *prints* `0 1 2 3 4`*, whereas* `count(5, False)` *prints* `4 3 2 1 0`*. In this way, your solution to this exercise may look like*

```
def traverse(G, start, depthFirst):
    if depthFirst:
        listType = # stack or queue (you figure out which!)
    else:
        listType = # the other one
    # same code for both depth-first and breadth-first
```

Exercise 5: Naive union-find

A *partition* of a set A is a family of non-empty subsets $(U_i)_{i \in I}$ of A such that:

- for all $i, j \in I$, $U_i \cap U_j = \emptyset$ (the sets are pairwise disjoint);
- $\bigcup_{i \in I} U_i = A$ (the sets cover A).

The sets U_i are called *equivalence classes*. The above conditions guarantee that each element of A is in exactly one equivalence class. We say that two elements of A are *equivalent* if they belong to the same equivalence class. Notice that, if $(U_i)_{i \in I}$ is a partition of A, then for all $i \neq j \in I$, defining $K := I \setminus \{j\}$ and, for all $k \in K$,

$$V_k := \begin{cases} U_k & \text{if } k \neq i, \\ U_i \cup U_j & \text{if } k = i, \end{cases}$$

yields another partition $(V_k)_{k \in K}$ of A. We say that such a partition results from *merging* the equivalence classes U_i and U_j. For simplicity, we will assume from now on that $A = \{0, 1, 2, \dots, n-1\}$ for some $n \in \mathbb{N}$.

A *union-find* structure (also known as *disjoint-set* structure) is a data structure used for efficiently manipulating partitions of a finite set A. In particular, union-find structures provide two fundamental functionalities:

- testing whether two elements of A are equivalent;
- merging two equivalence classes.

The basic idea is to pick, for each equivalence class U_i, an element of U_i called the *canonical representative* of the class, so that each element of A is associated with the canonical representative of its equivalence class. In this way, we may test whether a and b are equivalent by testing for equality of their canonical representatives, and we may merge two equivalence classes U_i, U_j by identifying the canonical representative of one to the canonical representative of the other.

Following the above description, a union-find structure provides two basic operations:

- `find(a)`, which returns the canonical representative of `a`;
- `union(a, b)`, which merges the equivalence classes of `a` and `b`.

A first implementation could work as follows: we represent the map taking each element `a` to its canonical representative as a list `l` such that `l[a]` is equal to the canonical representative of `a`. In this way:

- implementing `find(a)` is trivial: we simply return `l[a]`;
- implementing `union(a,b)` requires finding all elements in `l` which are equal to the canonical representative of `a` and replacing them with the canonical representative of `b` (or vice versa, the role of `a` and `b` is symmetric).

Complete the following Python class

```
class uf:
    def __init__(self, n):
        self.l = list(range(n))
    def find(self, a):
        # your code here
    def union(self, a, b):
        # your code here
    def toList(self):
        n = len(self.l)
        t = []
        for i in range(n):
            t.append([])
        for a in range(n):
            t[self.find(a)].append(a)
        r = []
        for i in range(n):
            if t[i] != []:
                r.append(t[i])
        return r
```

so that it implements union-find as described above. You may test your answer, for example, by running the following program:

```
m = uf(5)
m.union(0,4)
m.union(2,3)
print(m.toList())
```

which should print something like `[[1], [2, 3], [0, 4]]`. This is because the line `m = uf(5)` initializes `m` to represent the set $\{0,1,2,3,4\}$ partitioned into singletons, that is, the equivalence classes are $\{0\},\{1\},\{2\},\{3\},\{4\}$. After the first `union`, the partition becomes $\{1\},\{2\},\{3\},\{0,4\}$, and after the second union it becomes $\{1\},\{2,3\},\{0,4\}$. (NB: the exact order of the printed lists and of the elements within them may vary depending on the implementation; for example, the above code may print `[[0, 4], [1], [2, 3]]`, which obviously represents the same partition).

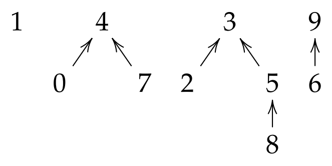
Exercise 6: Improving union-find

It is easy to analyze the complexity of union-find as implemented in the previous exercise. If n is the size of the set that we are partitioning, we have:

- `find` is $\Theta(1)$;
- `union` is $\Theta(n)$ (because we need to scan the whole list in order to update the canonical representatives).

It is possible to greatly improve the complexity of `union`, by applying two successive ideas.

The first idea is to represent each equivalence class as a tree whose root is the canonical representative. For example, the following forest

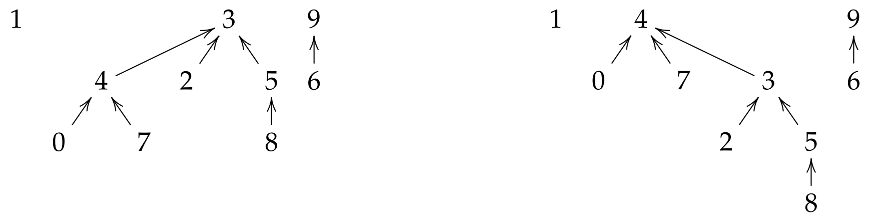


corresponds to the partition $\{1\},\{0,4,7\},\{2,3,5,8\},\{6,9\}$, with the canonical representative of each class being $1,4,3,9$, respectively. The correspondence is not one-to-one: the same partition yields more than one forest. For example, in the above forest, one may change 8 to point to 2, or even to the root 3 (trees need not be binary), without altering either the corresponding partition or the canonical representatives.

In this way, implementing `union(a, b)` may be done by:

- finding the root of the tree containing `a`, call it `p`;
- finding the root of the tree containing `b`, call it `q`;
- making either `p` point to `q`, or vice versa.

For example, if we execute `union(7, 2)` on the above partition, we get one of the following two forests, according to whether we make the root above 7 (which is 4) point to the root above 2 (which is 3), or vice versa:



Both forests of course correspond to the same partition, namely $\{1\}, \{0,2,3,4,5,7,8\}, \{6,9\}$. However, the forest on the left is more "compact" than the forest on the right, in the sense that its depth is smaller: it is of depth 2, whereas the forest on the right is of depth 3.

The above observation is where the second idea comes from: in order to keep the trees as shallow as possible, in a `union` operation we always make the shallow tree point towards the deep one (if they have the same depth, we choose arbitrarily). This has an important impact on the complexity of `find`: in fact, `find(a)` is now no longer a constant-time operation, because we need to walk up the edges of the forest in order to find the root above `a`. Since this operation requires a number of steps equal to the depth of `a` in its tree, limiting the depth of trees improves the efficiency of `find`. For example, in the above forest on the left, `find(8)` takes 2 steps, whereas the same operation takes 3 steps in the forest on the right.

It is possible to show that, with this careful `union` operation, the depth of trees is always bounded by $\log n$, where n is the size of the set being partitioned. So, summing up, we have:

- for `find(a)`, we look for the root above `a` in the forest, and we return it. This takes time linear in the depth of the forest, which we said above is bounded by $\log n$, so the `find` operation has complexity $\Theta(\log n)$.
- For `union(a, b)`, we need to call `find(a)` and `find(b)` to find the roots above `a` and `b`, respectively, and then we make the root of the shallow tree point to the root of the deep tree, which is a constant-time operation. Assuming that determining which tree is deeper may also be done in constant time (we will see how in a moment), the complexity of `union` is dominated by the two calls to `find`, and is therefore $\Theta(\log n)$ as well.

So we went from a constant-time `find` and linear `union`, to a logarithmic bound for both operations, which is a big improvement globally. Indeed, in the first case a finite sequence of `union` and `find` operations will have cost $O(n)$ (the cost is dominated by `union`, it does not help that `find` is constant-time), whereas in the second case the sequence will cost $O(\log n)$, an exponential improvement!

Implementation. A forest on n nodes may be implemented as a list `l` of length n such that `l[a]` is equal to the parent of `a` in the forest. For instance, the first forest introduced above will be represented by the list

```
[4, 1, 3, 3, 4, 3, 9, 4, 5, 9]
```

whereas the forest after the `union(7, 2)` operation will be represented by the list

```
[4, 1, 3, 3, 3, 3, 9, 4, 5, 9]
```

Observe that canonical representatives, that is, the roots of the forest, are those elements `a` such that `l[a] == a`. We may use this property to implement `find`.

For implementing `union`, we need to determine the depth of trees in the forest. Computing it on demand is too costly, so we use another list, call it `d`, such that `d[a]` contains the depth of the tree whose root is `a`. Initially, when every element is in its own equivalence class, `d` is set to zero everywhere. Subsequently, we only need to maintain the indices of `d` corresponding to the roots, because once a node `a` ceases to be a root, the value `d[a]` will never be looked at again. So a call to `union(a, b)` will:

- call `find(a)` and `find(b)`, obtaining `p` and `q`, respectively;
- look up `d[p]` and `d[q]` to see which one is smaller; say, for example, that `d[p] <= d[q]`;
- set `l[p] = q`, and increment `d[q]` by `1` in case `d[p] == d[q]` (because if `d[p] < d[q]`, then the depth of the tree whose root is `q` does not increase; make sure you understand why).

Write the above implementation in Python, completing the following class declaration:

```
class uf:
    def __init__(self, n):
        self.l = list(range(n))
        self.d = [0] * n
    def find(self, a):
        # your code here
    def union(self, a, b):
        # your code here
    def toList(self):
        n = len(self.l)
        t = []
        for i in range(n):
            t.append([])
        for a in range(n):
            t[self.find(a)].append(a)
        r = []
        for i in range(n):
            if t[i] != []:
                r.append(t[i])
        return r
```

You may test your implementation by running

```
m = uf(10)
m.union(0,4)
m.union(2,3)
m.union(3,5)
m.union(7,0)
m.union(6,9)
m.union(2,8)
print(m.toList())
print(m.l)
```

which should print something like

```
[[1], [2, 3, 5, 8], [0, 4, 7], [6, 9]]
[4, 1, 3, 3, 4, 3, 9, 4, 3, 9]
```

The first line shows the partition, which is the one corresponding to the first forest introduced above; the second line is the forest representation, which is not quite the one above: it is more shallow (its depth is only 1), as expected with an efficient implementation.

Exercise 7: Kruskal's algorithm

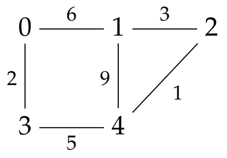
In the last lecture, we saw Kruskal's algorithm for finding the minimum spanning tree of a weighted undirected graph G:

1. initialize a forest F containing all nodes of G and no edge;
2. sort the edges of G by weight;
3. pick the edge e of minimum weight; if it adds no cycle to F, add e to F, otherwise discard e;
4. if there are no more edges, terminate, otherwise go back to 3.

Recall that the main problem in Kruskal's algorithm is to determine whether adding an edge to a forest makes it cyclic. Running an acyclicity test would be too costly, so we exploit the following observation. Given a forest F on the nodes $A = \{0, \dots, n-1\}$, its connected components form a partition of A. If the edge e we want to add is between i and j, we are certain that if i and j are not in the same equivalence class (that is, not in the same connected component), then adding e to F will add no cycle (convince yourself of this fact). Moreover, once we add e, the equivalence classes of i and j will be merged (that is, i and j will now be in the same connected component of F). So the two operations needed (testing whether two elements are equivalent and merging two equivalence classes) are precisely those offered by the union-find structure.

Write a Python function implementing Kruskal's algorithm, using the function `getEdges` of Exercise 2 to extract the weighted edges of the graph and the union-find structure of Exercise 6 to keep track of the connected components of the spanning forest.

Note that the minimum spanning tree is defined on weighted *undirected* graphs. For the purpose of this problem, we will represent such graphs with adjacency matrices `G` such that, for all `i`,`j`, at most one of `G[i][j]` and `G[j][i]` is not `-1`. If one of them is, then there is an undirected edge (of weight given by the entry) between `i` and `j`; if both are `-1`, then there is no edge between `i` and `j`. For example, the graph



may be represented by the "triangular" matrix

```
[[-1,  6, -1,  2, -1],
 [-1, -1,  3, -1,  9],
 [-1, -1, -1, -1,  1],
 [-1, -1, -1, -1,  5],
 [-1, -1, -1, -1, -1]]
```

To test your implementation, you may run it on the above graph, which should give you

```
[[-1, -1, -1,  2, -1],
 [-1, -1,  3, -1, -1],
 [-1, -1, -1, -1,  1],
 [-1, -1, -1, -1,  5],
 [-1, -1, -1, -1, -1]]
```

representing the tree