Partition Refinement

Atli Fannar Franklín

September 22, 2025

School of Computer Science

Reykjavík University

Opposite Union-Find

- In Union-Find we want to group things together more and more over time, keeping track of which items have been joined together.
- In certain situations we want quite the opposite, starting with everything in one group and breaking it up over time.
- This won't be the exact opposite behaviour, but close to it.
- This may seem rather specific, but it has quite a few uses and provides a good case study for us on a more niche data structure.
- It is useful for DFA minimisation, parallel scheduling and certain types of graph searches.

Split operation

- At the start our data structure should contain just a single group S of every element $1, 2, \ldots, n$.
- Then it should have a split operation. Split should take a set X as input and then split every group S in the data structure into $S \cap X$ and $S \setminus X$.
- For example if we start with $\{1, 2, 3, 4, 5\}$ and do `split(1, 2)` we should end up with $\{1, 2\}, \{3, 4, 5\}$.
- Next if we do `split(1, 4)` we get $\{1\}, \{2\}, \{3, 5\}, \{4\}$.
- Crucially we want this to be linear time in the number of input elements.

Group iteration

- The only other thing the data structure has to support is `group(x)`.
- This should return a list/vector/iterator over all the elements of the group x is in.
- Again this should take linear time in the number of elements returned.
- Together these operations define the partition refinement data structure.

Needed data

- So what info does our data structure need?

Needed data

- So what info does our data structure need?
    - We need an array that stores what group each element is in.

Needed data

- So what info does our data structure need?
    - We need an array that stores what group each element is in.
    - Then we need a data structure to store each group. These need to support constant time deletion from anywhere, so we use linked lists.

Needed data

- So what info does our data structure need?
  - We need an array that stores what group each element is in.
  - Then we need a data structure to store each group. These need to support constant time deletion from anywhere, so we use linked lists.
  - We still have to be able to find elements quickly, so we store a pointer to each element's linked list node in an array.

Needed data

- So what info does our data structure need?
    - We need an array that stores what group each element is in.
    - Then we need a data structure to store each group. These need to support constant time deletion from anywhere, so we use linked lists.
    - We still have to be able to find elements quickly, so we store a pointer to each element's linked list node in an array.
- Call the first `group_index`, the second `groups` and the last `locations`.

Group iteration

- With this data we can easily iterate over a group.
- Given an item x we find its group i as `group_index[x]`.
- Then we just find the group at `groups[i]` and iterate through that linked list.

Split operation

- The split operation is trickier. It's a bit easier if we allow ourselves a linearithmic time complexity, but it can be done in linear time.
- First we have to collect together elements in the input that are currently in the same group. We can do this by making a hash table that maps group indices to lists, then adding the elements in the input one by one (or by sorting).
- Then each of these lists should be a new group and we need to delete them from their current groups.

Split operation

- We then process these new groups one by one. Each time we add a new group with some new index j.
- For each element x we find their location in their linked list using `locations[x]` and delete that node.
- Then we add x to `groups[j]` and update `locations` to point there.
- Finally we set `group_index[x]` to j.

Bonus challenge

- This gives us a linear time complexity as desired.
- But linked lists are slow! They'll be fast enough for the homework, but as an added challenge one can try to implement all of this in the same time complexity using only vectors/arrays.