Union-Find data structure

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Programming, Data Structures and Algorithms using Python

Week 6

Kruskal's algoriththm for minimum cost spanning tree (MCST)

- Process edges in ascending order of cost
- If edge (u, v) does not create a cycle, add it
  - (u, v) can be added if u and v are in different components
  - Adding edge (u, v) merges these components
- How can we keep track of components and merge them efficiently?

Kruskal's algoriththm for minimum cost spanning tree (MCST)

- Process edges in ascending order of cost
- If edge (u, v) does not create a cycle, add it
  - (u, v) can be added if u and v are in different components
  - Adding edge (u, v) merges these components
- How can we keep track of components and merge them efficiently?

- Components partition vertices
  - Collection of disjoint sets
- Need data structure to maintain collection of disjoint sets
  - `find(v)` — return set containing `v`
  - `union(u,v)` — merge sets of `u`, `v`

Union-Find data strucrure

- A set S partitioned into components $\{C_1, C_2, \ldots, C_k\}$
  - Each $s \in S$ belongs to exactly one C_j

Union-Find data strucrure

- A set S partitioned into components $\{C_1, C_2, \ldots, C_k\}$
  - Each $s \in S$ belongs to exactly one C_j
- Support the following operations
  - `MakeUnionFind(S)` — set up initial singleton components $\{s\}$, for each $s \in S$
  - `Find(s)` — return the component containing s
  - `Union(s,s')` — merges components containing s, s'

Naive implementation

- Assume $S = \{0, 1, \ldots, n-1\}$

Naive implementation

- Assume $S = \{0, 1, \ldots, n-1\}$
- Set up a array/dictionary `Component`

Naive implementation

- Assume $S = \{0, 1, \ldots, n-1\}$
- Set up a array/dictionary `Component`
- `MakeUnionFind(S)` —
    - Set `Component[i] = i` for each `i`

Naive implementation

- Assume $S = \{0, 1, \dots, n-1\}$
- Set up a array/dictionary `Component`
- `MakeUnionFind(S)` —
  - Set `Component[i] = i` for each `i`
- `Find(i)`
  - Return `Component[i]`

Naive implementation

- Assume $S = \{0, 1, \ldots, n-1\}$
- Set up a array/dictionary `Component`
- `MakeUnionFind(S)` —
  - Set `Component[i] = i` for each `i`
- `Find(i)`
  - Return `Component[i]`
- `Union(i,j)`

```
c_old = Component[i]
c_new = Component[j]
for k in range(n):
    if Component[k] == c_old:
       Component[k] = c_new
```

Naive implementation

- Assume $S = \{0, 1, \dots, n-1\}$
- Set up a array/dictionary `Component`
- `MakeUnionFind(S)` —
  - Set `Component[i] = i` for each `i`
- `Find(i)`
  - Return `Component[i]`
- `Union(i,j)`

```
c_old = Component[i]
c_new = Component[j]
for k in range(n):
    if Component[k] == c_old:
        Component[k] = c_new
```

Complexity

- `MakeUnionFind(S)` — $O(n)$
- `Find(i)` — $O(1)$
- `Union(i,j)` — $O(n)$
- Sequence of m `Union()` operations takes time $O(mn)$

Improved implementation

- Another array/dictionary `Members`

Improved implementation

- Another array/dictionary `Members`
- For each component c, `Members[c]` is a list of its members
- `Size[c]` = `length(Members[c])` is the number of members

Improved implementation

- Another array/dictionary `Members`
- For each component c, `Members[c]` is a list of its members
- `Size[c]` $=$ `length(Members[c])` is the number of members

- `MakeUnionFind(S)`
  - Set `Component[i] = i` for all `i`
  - Set `Members[i] = [i]`, `Size[i] = 1` for all `i`

Improved implementation

- Another array/dictionary `Members`
- For each component c, `Members[c]` is a list of its members
- `Size[c]` $=$ `length(Members[c])` is the number of members

- `MakeUnionFind(S)`
  - Set `Component[i] = i` for all `i`
  - Set `Members[i] = [i]`, `Size[i] = 1` for all `i`
- `Find(i)`
  - Return `Component[i]`

Improved implementation

- Another array/dictionary `Members`
- For each component c, `Members[c]` is a list of its members
- `Size[c]` = `length(Members[c])` is the number of members

- `MakeUnionFind(S)`
  - Set `Component[i] = i` for all `i`
  - Set `Members[i] = [i]`, `Size[i] = 1` for all `i`
- `Find(i)`
  - Return `Component[i]`
- `Union(i,j)`

```
c_old = Component[i]
c_new = Component[j]
for k in Members[c_old]:
  Component[k] = c_new
  Members[c_new].append(k)
  Size[c_new] = Size[c_new] + 1
```

Why does this help?

- `MakeUnionFind(S)`
  - Set `Component[i] = i` for all `i`
  - Set `Members[i] = [i]`, `Size[i] = 1` for all `i`
- `Find(i)`
  - Return `Component[i]`
- `Union(i,j)`

```
c_old = Component[i]
c_new = Component[j]
for k in Members[c_old]:
    Component[k] = c_new
    Members[c_new].append(k)
    Size[c_new] = Size[c_new] + 1
```

Why does this help?

- `MakeUnionFind(S)`
  - Set `Component[i] = i` for all `i`
  - Set `Members[i] = [i]`, `Size[i] = 1` for all `i`
- `Find(i)`
  - Return `Component[i]`
- `Union(i,j)`

```
c_old = Component[i]
c_new = Component[j]
for k in Members[c_old]:
    Component[k] = c_new
    Members[c_new].append(k)
    Size[c_new] = Size[c_new] + 1
```

- `Members[c_old]` allows us to merge `Component[i]` into `Component[j]` in time $O(Size[c\_old])$ rather than $O(n)$

Why does this help?

- `MakeUnionFind(S)`
  - Set `Component[i] = i` for all `i`
  - Set `Members[i] = [i]`, `Size[i] = 1` for all `i`
- `Find(i)`
  - Return `Component[i]`
- `Union(i,j)`

```
c_old = Component[i]
c_new = Component[j]
for k in Members[c_old]:
   Component[k] = c_new
   Members[c_new].append(k)
   Size[c_new] = Size[c_new] + 1
```

- `Members[c_old]` allows us to merge `Component[i]` into `Component[j]` in time $O(Size[c\_old])$ rather than $O(n)$
- How can we make use of `Size[c]`
  - Always merge smaller component into larger one
  - If `Size[c] < Size[c']` relabel `c` as `c'`, else relabel `c'` as `c`

Why does this help?

- `MakeUnionFind(S)`
  - Set `Component[i] = i` for all `i`
  - Set `Members[i] = [i]`, `Size[i] = 1` for all `i`
- `Find(i)`
  - Return `Component[i]`
- `Union(i,j)`

```
c_old = Component[i]
c_new = Component[j]
for k in Members[c_old]:
    Component[k] = c_new
    Members[c_new].append(k)
    Size[c_new] = Size[c_new] + 1
```

- `Members[c_old]` allows us to merge `Component[i]` into `Component[j]` in time $O(Size[c\_old])$ rather than $O(n)$
- How can we make use of `Size[c]`
  - Always merge smaller component into larger one
  - If `Size[c] < Size[c']` relabel `c` as `c'`, else relabel `c'` as `c`
- Individual merge operations can still take time $O(n)$
  - Both `Size[c]`, `Size[c']` could be about $n/2$
  - More careful accounting

Why does this help?

- `MakeUnionFind(S)`
  - Set `Component[i] = i` for all `i`
  - Set `Members[i] = [i]`, `Size[i] = 1` for all `i`
- `Find(i)`
  - Return `Component[i]`
- `Union(i,j)`

```
c_old = Component[i]
c_new = Component[j]
for k in Members[c_old]:
    Component[k] = c_new
    Members[c_new].append(k)
    Size[c_new] = Size[c_new] + 1
```

- Always merge smaller component into larger one

Why does this help?

- `MakeUnionFind(S)`
  - Set `Component[i] = i` for all `i`
  - Set `Members[i] = [i]`, `Size[i] = 1` for all `i`
- `Find(i)`
  - Return `Component[i]`
- `Union(i,j)`

```
c_old = Component[i]
c_new = Component[j]
for k in Members[c_old]:
    Component[k] = c_new
    Members[c_new].append(k)
    Size[c_new] = Size[c_new] + 1
```

- Always merge smaller component into larger one
- For each `i`, size of `Component[i]` at least doubles each time it is relabelled

Why does this help?

- `MakeUnionFind(S)`
  - Set `Component[i] = i` for all `i`
  - Set `Members[i] = [i]`, `Size[i] = 1` for all `i`
- `Find(i)`
  - Return `Component[i]`
- `Union(i,j)`

```
c_old = Component[i]
c_new = Component[j]
for k in Members[c_old]:
    Component[k] = c_new
    Members[c_new].append(k)
    Size[c_new] = Size[c_new] + 1
```

- Always merge smaller component into larger one
- For each `i`, size of `Component[i]` at least doubles each time it is relabelled
- After m `Union()` operations, at most $2m$ elements have been "touched"
  - Size of `Component[i]` is at most $2m$

Why does this help?

- `MakeUnionFind(S)`
  - Set `Component[i] = i` for all `i`
  - Set `Members[i] = [i]`, `Size[i] = 1` for all `i`
- `Find(i)`
  - Return `Component[i]`
- `Union(i,j)`

```
c_old = Component[i]
c_new = Component[j]
for k in Members[c_old]:
   Component[k] = c_new
   Members[c_new].append(k)
   Size[c_new] = Size[c_new] + 1
```

- Always merge smaller component into larger one
- For each `i`, size of `Component[i]` at least doubles each time it is relabelled
- After m `Union()` operations, at most $2m$ elements have been "touched"
  - Size of `Component[i]` is at most $2m$
- Size of `Component[i]` grows as $1, 2, 4, \ldots$, so `i` changes component at most $\log m$ times

Why does this help?

- Always merge smaller component into larger one
- For each `i`, size of `Component[i]` at least doubles each time it is relabelled
- After m `Union()` operations, at most $2m$ elements have been "touched"
  - Size of `Component[i]` is at most $2m$
- Size of `Component[i]` grows as $1, 2, 4, \ldots$, so `i` changes component at most $\log m$ times

Why does this help?

- Always merge smaller component into larger one
- For each `i`, size of `Component[i]` at least doubles each time it is relabelled
- After m `Union()` operations, at most $2m$ elements have been "touched"
  - Size of `Component[i]` is at most $2m$
- Size of `Component[i]` grows as $1, 2, 4, \ldots$, so `i` changes component at most $\log m$ times

- Over m updates
  - At most $2m$ elements are relabelled
  - Each one at most $O(\log m)$ times

Why does this help?

- Always merge smaller component into larger one
- For each `i`, size of `Component[i]` at least doubles each time it is relabelled
- After m `Union()` operations, at most $2m$ elements have been "touched"
    - Size of `Component[i]` is at most $2m$
- Size of `Component[i]` grows as $1, 2, 4, \ldots$, so `i` changes component at most $\log m$ times

- Over m updates
    - At most $2m$ elements are relabelled
    - Each one at most $O(\log m)$ times
- Overall, m `Union()` operations take time $O(m \log m)$

Why does this help?

- Always merge smaller component into larger one
- For each `i`, size of `Component[i]` at least doubles each time it is relabelled
- After m `Union()` operations, at most $2m$ elements have been "touched"
  - Size of `Component[i]` is at most $2m$
- Size of `Component[i]` grows as $1, 2, 4, \ldots$, so `i` changes component at most $\log m$ times

- Over m updates
  - At most $2m$ elements are relabelled
  - Each one at most $O(\log m)$ times
- Overall, m `Union()` operations take time $O(m \log m)$
- Works out to time $O(\log m)$ per `Union()` operation
  - Amortised complexity of `Union()` is $O(\log m)$

Back to Kruskal's algorithm

- Sort $E = \{e_0, e_1, \ldots, e_{m-1}\}$ in ascending order

Back to Kruskal's algorithm

- Sort $E = \{e_0, e_1, \ldots, e_{m-1}\}$ in ascending order
- `MakeUnionFind(V)` — each vertex `j` is in component `j`

Back to Kruskal's algorithm

- Sort $E = \{e_0, e_1, \ldots, e_{m-1}\}$ in ascending order
- `MakeUnionFind(V)` — each vertex `j` is in component `j`
- Adding and edge $e_k = (u, v)$ to the tree
  - Check that `Find(u) != Find(v)`
  - Merge components: `Union(Component[u],Component[v])`

Back to Kruskal's algorithm

- Sort $E = \{e_0, e_1, \ldots, e_{m-1}\}$ in ascending order
- `MakeUnionFind(V)` — each vertex `j` is in component `j`
- Adding and edge $e_k = (u, v)$ to the tree
  - Check that `Find(u) != Find(v)`
  - Merge components: `Union(Component[u],Component[v])`
- Tree has $n-1$ edges, so $O(n)$ `Union()` operations
  - $O(n \log n)$ amortised cost, overall

Back to Kruskal's algorithm

- Sort $E = \{e_0, e_1, \ldots, e_{m-1}\}$ in ascending order
- `MakeUnionFind(V)` — each vertex `j` is in component `j`
- Adding and edge $e_k = (u, v)$ to the tree
  - Check that `Find(u) != Find(v)`
  - Merge components: `Union(Component[u],Component[v])`
- Tree has $n-1$ edges, so $O(n)$ `Union()` operations
  - $O(n \log n)$ amortised cost, overall
- Sorting E takes $O(m \log m)$
  - Equivalently $O(m \log n)$, since $m \leq n^2$

Back to Kruskal's algorithm

- Sort $E = \{e_0, e_1, \ldots, e_{m-1}\}$ in ascending order
- `MakeUnionFind(V)` — each vertex `j` is in component `j`
- Adding and edge $e_k = (u, v)$ to the tree
  - Check that `Find(u) != Find(v)`
  - Merge components: `Union(Component[u],Component[v])`
- Tree has $n-1$ edges, so $O(n)$ `Union()` operations
  - $O(n \log n)$ amortised cost, overall
- Sorting E takes $O(m \log m)$
  - Equivalently $O(m \log n)$, since $m \leq n^2$
- Overall time, $O((m+n) \log n)$

Summary

- Implement Union-Find using arrays/dictionaries `Component`, `Member`, `Size`
    - `MakeUnionFind(S)` is $O(n)$
    - `Find(i)` is $O(1)$
    - Across m operations, amortised complexity of each `Union()` operation is $\log m$

Summary

- Implement Union-Find using arrays/dictionaries `Component`, `Member`, `Size`
  - `MakeUnionFind(S)` is $O(n)$
  - `Find(i)` is $O(1)$
  - Across m operations, amortised complexity of each `Union()` operation is $\log m$
- Can also maintain `Members[k]` as a tree rather than as a list
  - `Union()` becomes $O(1)$
  - With clever updates to the tree, `Find()` has amortised complexity very close to $O(1)$