

CMPSCI 611: Advanced Algorithms

Lecture 7: Union Find and Bipartite Matching

Andrew McGregor

Last Compiled: October 20, 2009

Outline

Union-Find Data Structure

Recall Kruskal's Algorithm. . .

Problem: Given an undirected, connected graph $G = (V, E)$ with positive edge weights, find the minimum-weight subset $E' \subset E$ such that the graph $G = (V, E')$ is a minimum spanning tree.

Recall Kruskal's Algorithm. . .

Problem: Given an undirected, connected graph $G = (V, E)$ with positive edge weights, find the minimum-weight subset $E' \subset E$ such that the graph $G = (V, E')$ is a minimum spanning tree.

Algorithm (Kruskal)

1. *Sort edges by non-decreasing weight*
2. $F = \emptyset$
3. *Until F is a spanning tree of G*
 - 3.1 *Get the next edge e*
 - 3.2 *If $F + e$ is acyclic then $F = F + e$*

Recall Kruskal's Algorithm. . .

Problem: Given an undirected, connected graph $G = (V, E)$ with positive edge weights, find the minimum-weight subset $E' \subset E$ such that the graph $G = (V, E')$ is a minimum spanning tree.

Algorithm (Kruskal)

1. *Sort edges by non-decreasing weight*
2. $F = \emptyset$
3. *Until F is a spanning tree of G*
 - 3.1 *Get the next edge e*
 - 3.2 *If $F + e$ is acyclic then $F = F + e$*

Easy to get $O(|E| \log |E| + |V|^2)$ running time. Let's see how to improve to $O(|E| \log |E|)$ via the union-find data structure.

Union-Find Data Structure

Encodes a set of disjoint sets where each set contains an element designated as the “label” of the set. E.g.,

$\{a, b, c\}$ labeled “a” $\{d, e, f\}$ labeled “e”

Union-Find Data Structure

Encodes a set of disjoint sets where each set contains an element designated as the “label” of the set. E.g.,

$\{a, b, c\}$ labeled “a” $\{d, e, f\}$ labeled “e”

Supports three operations:

1. **Make-Set(v)**: Adds a set $\{v\}$ with label “v”

Union-Find Data Structure

Encodes a set of disjoint sets where each set contains an element designated as the “label” of the set. E.g.,

$\{a, b, c\}$ labeled “a” $\{d, e, f\}$ labeled “e”

Supports three operations:

1. **Make-Set(v)**: Adds a set $\{v\}$ with label “v”

$\{a, b, c\}$ labeled “a” $\{d, e, f\}$ labeled “e” $\{v\}$ labeled “v”

Union-Find Data Structure

Encodes a set of disjoint sets where each set contains an element designated as the “label” of the set. E.g.,

$\{a, b, c\}$ labeled “a” $\{d, e, f\}$ labeled “e”

Supports three operations:

1. **Make-Set(v)**: Adds a set $\{v\}$ with label “v”

$\{a, b, c\}$ labeled “a” $\{d, e, f\}$ labeled “e” $\{v\}$ labeled “v”

2. **Union-Set(u, v)**: Replaces sets including u and v with a new set that is union of both sets and labels this set by some element it contains.

Union-Find Data Structure

Encodes a set of disjoint sets where each set contains an element designated as the “label” of the set. E.g.,

$\{a, b, c\}$ labeled “a” $\{d, e, f\}$ labeled “e”

Supports three operations:

1. **Make-Set(v)**: Adds a set $\{v\}$ with label “v”

$\{a, b, c\}$ labeled “a” $\{d, e, f\}$ labeled “e” $\{v\}$ labeled “v”

2. **Union-Set(u, v)**: Replaces sets including u and v with a new set that is union of both sets and labels this set by some element it contains. E.g., $\text{Union-Set}(f, v)$ yields:

$\{a, b, c\}$ labeled “a” $\{d, e, f, v\}$ labeled “e”

3. **Find(v)**: Returns the label of the set including v

Kruskal's Algorithm with Union-Find

Algorithm (Kruskal)

1. *Sort edges by non-decreasing weight*
2. *For each vertex $v \in V$: Make-Set(v)*
3. $F = \emptyset$
4. *For each edge $e = (u, v)$ in E*
 - 4.1 *If $\text{Find}(u) \neq \text{Find}(v)$ then $\text{Union}(u, v)$ and $F = F + e$*

Kruskal's Algorithm with Union-Find

Algorithm (Kruskal)

1. *Sort edges by non-decreasing weight*
2. *For each vertex $v \in V$: Make-Set(v)*
3. $F = \emptyset$
4. *For each edge $e = (u, v)$ in E*
 - 4.1 *If Find(u) \neq Find(v) then Union(u, v) and $F = F + e$*

Well, how should we implement union-find...

Simple Implementation of Union-Find

1. Each disjoint set is stored as a linked list of nodes

Simple Implementation of Union-Find

1. Each disjoint set is stored as a linked list of nodes
2. Each node consists of three data items:

Simple Implementation of Union-Find

1. Each disjoint set is stored as a linked list of nodes
2. Each node consists of three data items:
 - 2.1 name of element

Simple Implementation of Union-Find

1. Each disjoint set is stored as a linked list of nodes
2. Each node consists of three data items:
 - 2.1 name of element
 - 2.2 "label" pointer to label of the set

Simple Implementation of Union-Find

1. Each disjoint set is stored as a linked list of nodes
2. Each node consists of three data items:
 - 2.1 name of element
 - 2.2 “label” pointer to label of the set
 - 2.3 “next” pointer to next node in list

Simple Implementation of Union-Find

1. Each disjoint set is stored as a linked list of nodes
2. Each node consists of three data items:
 - 2.1 name of element
 - 2.2 “label” pointer to label of the set
 - 2.3 “next” pointer to next node in list
3. Also maintain auxiliary pointer for each label to last node of corresponding list and the size of this list.

Simple Implementation of Union-Find

1. Each disjoint set is stored as a linked list of nodes
 2. Each node consists of three data items:
 - 2.1 name of element
 - 2.2 "label" pointer to label of the set
 - 2.3 "next" pointer to next node in list
 3. Also maintain auxiliary pointer for each label to last node of corresponding list and the size of this list.
-
1. **Make-Set(v)**: Takes $O(1)$ time to add a single node.

Simple Implementation of Union-Find

1. Each disjoint set is stored as a linked list of nodes
 2. Each node consists of three data items:
 - 2.1 name of element
 - 2.2 "label" pointer to label of the set
 - 2.3 "next" pointer to next node in list
 3. Also maintain auxiliary pointer for each label to last node of corresponding list and the size of this list.
-
1. **Make-Set(v)**: Takes $O(1)$ time to add a single node.
 2. **Find(v)**: Takes $O(1)$ time to follow pointer to label.

Simple Implementation of Union-Find

1. Each disjoint set is stored as a linked list of nodes
 2. Each node consists of three data items:
 - 2.1 name of element
 - 2.2 "label" pointer to label of the set
 - 2.3 "next" pointer to next node in list
 3. Also maintain auxiliary pointer for each label to last node of corresponding list and the size of this list.
-
1. **Make-Set(v)**: Takes $O(1)$ time to add a single node.
 2. **Find(v)**: Takes $O(1)$ time to follow pointer to label.
 3. **Union-Set(u, v)**: $O(\text{size of smaller set})$.

Simple Implementation of Union-Find

1. Each disjoint set is stored as a linked list of nodes
 2. Each node consists of three data items:
 - 2.1 name of element
 - 2.2 "label" pointer to label of the set
 - 2.3 "next" pointer to next node in list
 3. Also maintain auxiliary pointer for each label to last node of corresponding list and the size of this list.
-
1. **Make-Set(v)**: Takes $O(1)$ time to add a single node.
 2. **Find(v)**: Takes $O(1)$ time to follow pointer to label.
 3. **Union-Set(u, v)**: $O(\text{size of smaller set})$.
 - ▶ Update "next" pointer at end of longer list to point to start of shorter list

Simple Implementation of Union-Find

1. Each disjoint set is stored as a linked list of nodes
 2. Each node consists of three data items:
 - 2.1 name of element
 - 2.2 "label" pointer to label of the set
 - 2.3 "next" pointer to next node in list
 3. Also maintain auxiliary pointer for each label to last node of corresponding list and the size of this list.
-
1. **Make-Set(v)**: Takes $O(1)$ time to add a single node.
 2. **Find(v)**: Takes $O(1)$ time to follow pointer to label.
 3. **Union-Set(u, v)**: $O(\text{size of smaller set})$.
 - ▶ Update "next" pointer at end of longer list to point to start of shorter list
 - ▶ Update "label" pointers of shorter list to point to label of other list

Simple Implementation of Union-Find

1. Each disjoint set is stored as a linked list of nodes
 2. Each node consists of three data items:
 - 2.1 name of element
 - 2.2 "label" pointer to label of the set
 - 2.3 "next" pointer to next node in list
 3. Also maintain auxiliary pointer for each label to last node of corresponding list and the size of this list.
-
1. **Make-Set(v)**: Takes $O(1)$ time to add a single node.
 2. **Find(v)**: Takes $O(1)$ time to follow pointer to label.
 3. **Union-Set(u, v)**: $O(\text{size of smaller set})$.
 - ▶ Update "next" pointer at end of longer list to point to start of shorter list
 - ▶ Update "label" pointers of shorter list to point to label of other list
 - ▶ Update auxiliary pointers and size information

Union-Find Analysis

Theorem

Consider a sequence of m operations including n Make-Set operations. Total running time is $O(m + n \log n)$.

Union-Find Analysis

Theorem

Consider a sequence of m operations including n Make-Set operations. Total running time is $O(m + n \log n)$.

Proof.

- ▶ Total time from Find and Make-Set: $O(n + m)$

Union-Find Analysis

Theorem

Consider a sequence of m operations including n Make-Set operations. Total running time is $O(m + n \log n)$.

Proof.

- ▶ Total time from Find and Make-Set: $O(n + m)$
- ▶ Total time from Union: $O(n \log n)$

Union-Find Analysis

Theorem

Consider a sequence of m operations including n Make-Set operations. Total running time is $O(m + n \log n)$.

Proof.

- ▶ Total time from Find and Make-Set: $O(n + m)$
- ▶ Total time from Union: $O(n \log n)$
 - ▶ Updating auxiliary pointers and next pointers: $O(n)$

Union-Find Analysis

Theorem

Consider a sequence of m operations including n Make-Set operations. Total running time is $O(m + n \log n)$.

Proof.

- ▶ Total time from Find and Make-Set: $O(n + m)$
- ▶ Total time from Union: $O(n \log n)$
 - ▶ Updating auxiliary pointers and next pointers: $O(n)$
 - ▶ Updating label pointers: $O(n \log n)$

Union-Find Analysis

Theorem

Consider a sequence of m operations including n Make-Set operations. Total running time is $O(m + n \log n)$.

Proof.

- ▶ Total time from Find and Make-Set: $O(n + m)$
- ▶ Total time from Union: $O(n \log n)$
 - ▶ Updating auxiliary pointers and next pointers: $O(n)$
 - ▶ Updating label pointers: $O(n \log n)$ because the label pointer for a node can be updated at most $\log_2 n$ times.



Hence, Kruskal's algorithm can be implemented in time

$$O(|E| \log |E|) + O(|V| + |E| \log |E|) = O(|E| \log |E|)$$

Faster Implementation of Union Find

Theorem

There exists an implementation that, given a sequence of n Make-Set operations and m total operations, takes $O(m\alpha(n))$ time where α is the inverse Ackermann's function.

Faster Implementation of Union Find

Theorem

There exists an implementation that, given a sequence of n Make-Set operations and m total operations, takes $O(m\alpha(n))$ time where α is the inverse Ackermann's function.

Definition (Ackermann's Function)

Define a sequence of functions: $A_0(x) = 1 + x$ and

$$A_k(x) = A_{k-1}(A_{k-1}(\dots A_{k-1}(x) \dots))$$

where A_{k-1} is applied x times. The Ackermann function is defined as $A(k) = A_k(2)$. $\alpha(n)$ is defined as smallest k such that $A(k) \geq n$.

Faster Implementation of Union Find

Theorem

There exists an implementation that, given a sequence of n Make-Set operations and m total operations, takes $O(m\alpha(n))$ time where α is the inverse Ackermann's function.

Definition (Ackermann's Function)

Define a sequence of functions: $A_0(x) = 1 + x$ and

$$A_k(x) = A_{k-1}(A_{k-1}(\dots A_{k-1}(x) \dots))$$

where A_{k-1} is applied x times. The Ackermann function is defined as $A(k) = A_k(2)$. $\alpha(n)$ is defined as smallest k such that $A(k) \geq n$.

Example

$\alpha(n) \leq 4$ for all $n \leq 2^{2^{2^{(\dots 2)}}}$ where tower is of height 2048.

Idea Behind Faster Implementation

- ▶ Store each set as a rooted tree.

Idea Behind Faster Implementation

- ▶ Store each set as a rooted tree.
- ▶ Each node encodes an element and pointer to the parent.

Idea Behind Faster Implementation

- ▶ Store each set as a rooted tree.
- ▶ Each node encodes an element and pointer to the parent.
- ▶ The element at the root is the label of the set.

Idea Behind Faster Implementation

- ▶ Store each set as a rooted tree.
 - ▶ Each node encodes an element and pointer to the parent.
 - ▶ The element at the root is the label of the set.
1. **Make-Set(v)**: Takes $O(1)$ time to add a single node.

Idea Behind Faster Implementation

- ▶ Store each set as a rooted tree.
 - ▶ Each node encodes an element and pointer to the parent.
 - ▶ The element at the root is the label of the set.
1. **Make-Set(v)**: Takes $O(1)$ time to add a single node.
 2. **Find(v)**: Takes $O(d_v)$ time where d_v is the depth of v

Idea Behind Faster Implementation

- ▶ Store each set as a rooted tree.
 - ▶ Each node encodes an element and pointer to the parent.
 - ▶ The element at the root is the label of the set.
1. **Make-Set(v)**: Takes $O(1)$ time to add a single node.
 2. **Find(v)**: Takes $O(d_v)$ time where d_v is the depth of v
 3. **Union-Set(u, v)**: $O(d_v + d_u)$ time

Idea Behind Faster Implementation

- ▶ Store each set as a rooted tree.
 - ▶ Each node encodes an element and pointer to the parent.
 - ▶ The element at the root is the label of the set.
1. **Make-Set(v)**: Takes $O(1)$ time to add a single node.
 2. **Find(v)**: Takes $O(d_v)$ time where d_v is the depth of v
 3. **Union-Set(u, v)**: $O(d_v + d_u)$ time
 - ▶ Perform Find(u), and Find(v)

Idea Behind Faster Implementation

- ▶ Store each set as a rooted tree.
 - ▶ Each node encodes an element and pointer to the parent.
 - ▶ The element at the root is the label of the set.
1. **Make-Set(v)**: Takes $O(1)$ time to add a single node.
 2. **Find(v)**: Takes $O(d_v)$ time where d_v is the depth of v
 3. **Union-Set(u, v)**: $O(d_v + d_u)$ time
 - ▶ Perform Find(u), and Find(v)
 - ▶ Add pointer from root of smaller tree to root of larger tree

Idea Behind Faster Implementation

- ▶ Store each set as a rooted tree.
 - ▶ Each node encodes an element and pointer to the parent.
 - ▶ The element at the root is the label of the set.
1. **Make-Set(v)**: Takes $O(1)$ time to add a single node.
 2. **Find(v)**: Takes $O(d_v)$ time where d_v is the depth of v
 3. **Union-Set(u, v)**: $O(d_v + d_u)$ time
 - ▶ Perform Find(u), and Find(v)
 - ▶ Add pointer from root of smaller tree to root of larger tree

Extra Trick! Do *path compression*. Whenever we perform a Find operation, update the pointers from all nodes encountered to point to the root. Only increases time by a constant factor but saves time for future Find operations.