5.3: Disjoint Sets

Frank Stajano

Thomas Sauerwald

Lent 2015

Disjoint Sets (aka Union Find)

Handle makeSet(Item x)
Precondition: none of the existing sets contains x
Behaviour: create a new set \{x\} and return its handle

Handle findSet(Item x)
Precondition: there exists a set that contains x (given pointer to x)
Behaviour: return the handle of the set that contains x

Handle union(Handle h, Handle g)
Precondition: h \neq g
Behaviour: merge two disjoint sets and return handle of new set

Disjoint Sets Data Structure

\[
h_0 = \text{makeSet}(x)
\]
\[
h_0 \rightarrow h_1 = \text{findSet}(y)
\]
\[
h_1 \rightarrow h_4 = \text{union}(h_0, h_3)
\]
\[
h_5 = \text{union}(h_1, h_2)
\]
\[
y
\]
Outline

Disjoint Sets

Introduction to Graphs and Graph Searching
Disjoint Sets (aka Union Find)

Handle MakeSet(Item x)
Precondition: none of the existing sets contains x
Behaviour: create a new set \{x\} and return its handle

Handle FindSet(Item x)
Precondition: there exists a set that contains x (given pointer to x)
Behaviour: return the handle of the set that contains x

Handle Union(Handle h, Handle g)
Precondition: h \neq g
Behaviour: merge two disjoint sets and return handle of new set

Disjoint Sets Data Structure

\[
\begin{align*}
&x_0 = \text{makeSet}(x) \\
&x_1 = \text{findSet}(y) \\
&x_4 = \text{Union}(x_0, x_3) \\
&x_5 = \text{Union}(x_1, x_2)
\end{align*}
\]
Disjoint Sets (aka Union Find)

Disjoint Sets Data Structure

- **Handle MakeSet(Item x)**
  
  Precondition: none of the existing sets contains x
  
  Behaviour: create a new set \{x\} and return its handle

\[h_0 = \text{makeSet}(x)\]
Disjoint Sets (aka Union Find)

Disjoint Sets Data Structure

- **Handle MakeSet(Item x)**
  
  Precondition: none of the existing sets contains \( x \)
  
  Behaviour: create a new set \( \{x\} \) and return its handle

\[
h_0 = \text{makeSet}(x)
\]
Disjoint Sets Data Structure

- **Handle `MakeSet(Item x)`**
  - Precondition: none of the existing sets contains `x`
  - Behaviour: create a new set `{x}` and return its handle

\[
h_0 = \text{makeSet}(x)
\]
Disjoint Sets (aka Union Find)

**Disjoint Sets Data Structure**

- **Handle MakeSet(Item x)**
  - Precondition: none of the existing sets contains x
  - Behaviour: create a new set \{x\} and return its handle

- **Handle FindSet(Item x)**
  - Precondition: there exists a set that contains x (given pointer to x)
  - Behaviour: return the handle of the set that contains x

![Diagram of disjoint sets]

Disjoint Sets Data Structure

\[
\begin{align*}
\text{h}_0 &= \text{makeSet}(x) \\
\text{h}_1 &= \text{findSet}(y) \\
\text{h}_4 &= \text{Union} (\text{h}_0, \text{h}_3) \\
\text{h}_5 &= \text{Union} (\text{h}_1, \text{h}_2) \\
\end{align*}
\]
Disjoint Sets (aka Union Find)

**Disjoint Sets Data Structure**

- **Handle MakeSet(Item x)**
  - Precondition: none of the existing sets contains \( x \)
  - Behaviour: create a new set \{x\} and return its handle

- **Handle FindSet(Item x)**
  - Precondition: there exists a set that contains \( x \) (given pointer to \( x \))
  - Behaviour: return the handle of the set that contains \( x \)

\[
\begin{align*}
  h_0 &= \text{MakeSet}(x) \\
  h_1 &= \text{FindSet}(y) \\
  h_4 &= \text{Union}(h_0, h_3) \\
  h_5 &= \text{Union}(h_1, h_2)
\end{align*}
\]
Disjoint Sets (aka Union Find)

Disjoint Sets Data Structure

- **Handle `MakeSet(Item x)`**
  - Precondition: none of the existing sets contains \( x \)
  - Behaviour: create a new set \{\( x \)\} and return its handle

- **Handle `FindSet(Item x)`**
  - Precondition: there exists a set that contains \( x \) (given pointer to \( x \))
  - Behaviour: return the handle of the set that contains \( x \)

\[
h_1 = \text{findSet}(y)
\]
Disjoint Sets (aka Union Find)

Disjoint Sets Data Structure

- **Handle MakeSet(Item x)**
  - Precondition: none of the existing sets contains \( x \)
  - Behaviour: create a new set \( \{x\} \) and return its handle

- **Handle FindSet(Item x)**
  - Precondition: there exists a set that contains \( x \) (given pointer to \( x \))
  - Behaviour: return the handle of the set that contains \( x \)

- **Handle Union(Handle h, Handle g)**
  - Precondition: \( h \neq g \)
  - Behaviour: merge two disjoint sets and return handle of new set

\[
\begin{align*}
h_0 &= \text{makeSet}(x) \\
h_1 &= \text{findSet}(y) \\
h_4 &= \text{Union}(h_0, h_3) \\
h_5 &= \text{Union}(h_1, h_2)
\end{align*}
\]
Disjoint Sets (aka Union Find)

Disjoint Sets Data Structure

- **Handle MakeSet(Item x)**
  - Precondition: none of the existing sets contains $x$
  - Behaviour: create a new set $\{x\}$ and return its handle

- **Handle FindSet(Item x)**
  - Precondition: there exists a set that contains $x$ (given pointer to $x$)
  - Behaviour: return the handle of the set that contains $x$

- **Handle Union(Handle h, Handle g)**
  - Precondition: $h \neq g$
  - Behaviour: merge two disjoint sets and return handle of new set

$h_4 = \text{Union}(h_0, h_3)$
Disjoint Sets (aka Union Find)

- **Handle MakeSet(Item x)**
  - Precondition: none of the existing sets contains x
  - Behaviour: create a new set \( \{x\} \) and return its handle

- **Handle FindSet(Item x)**
  - Precondition: there exists a set that contains x (given pointer to x)
  - Behaviour: return the handle of the set that contains x

- **Handle Union(Handle h, Handle g)**
  - Precondition: \( h \neq g \)
  - Behaviour: merge two disjoint sets and return handle of new set

```
\[ h_4 = \text{Union}(h_0, h_3) \]
```
Disjoint Sets (aka Union Find)

- **Handle MakeSet(Item x)**
  Precondition: none of the existing sets contains x
  Behaviour: create a new set \{x\} and return its handle

- **Handle FindSet(Item x)**
  Precondition: there exists a set that contains x (given pointer to x)
  Behaviour: return the handle of the set that contains x

- **Handle Union(Handle h, Handle g)**
  Precondition: \( h \neq g \)
  Behaviour: merge two disjoint sets and return handle of new set

\[
h_4 = \text{Union}(h_0, h_3)
\]
Disjoint Sets (aka Union Find)

**Disjoint Sets Data Structure**

- **Handle MakeSet(Item x)**
  - **Precondition:** none of the existing sets contains $x$
  - **Behaviour:** create a new set $\{x\}$ and return its handle

- **Handle FindSet(Item x)**
  - **Precondition:** there exists a set that contains $x$ (given pointer to $x$)
  - **Behaviour:** return the handle of the set that contains $x$

- **Handle Union(Handle h, Handle g)**
  - **Precondition:** $h \neq g$
  - **Behaviour:** merge two disjoint sets and return handle of new set

$h_4 = \text{Union}(h_0, h_3)$
Disjoint Sets (aka Union Find)

Disjoint Sets Data Structure

- **Handle MakeSet(Item x)**
  Precondition: none of the existing sets contains \( x \)
  Behaviour: create a new set \( \{x\} \) and return its handle

- **Handle FindSet(Item x)**
  Precondition: there exists a set that contains \( x \) (given pointer to \( x \))
  Behaviour: return the handle of the set that contains \( x \)

- **Handle Union(Handle h, Handle g)**
  Precondition: \( h \neq g \)
  Behaviour: merge two disjoint sets and return handle of new set

![Diagram of disjoint sets]

5.3: Disjoint Sets
Disjoint Sets (aka Union Find)

- **Handle MakeSet(Item x)**
  - Precondition: none of the existing sets contains x
  - Behaviour: create a new set \{x\} and return its handle

- **Handle FindSet(Item x)**
  - Precondition: there exists a set that contains x (given pointer to x)
  - Behaviour: return the handle of the set that contains x

- **Handle Union(Handle h, Handle g)**
  - Precondition: \( h \neq g \)
  - Behaviour: merge two disjoint sets and return handle of new set

\[ h_5 = \text{Union}(h_1, h_2) \]
Disjoint Sets (aka Union Find)

- **Handle MakeSet(Item x)**
  
  **Precondition:** none of the existing sets contains x
  
  **Behaviour:** create a new set \{x\} and return its handle

- **Handle FindSet(Item x)**
  
  **Precondition:** there exists a set that contains x (given pointer to x)
  
  **Behaviour:** return the handle of the set that contains x

- **Handle Union(Handle h, Handle g)**
  
  **Precondition:** \( h \neq g \)
  
  **Behaviour:** merge two disjoint sets and return handle of new set

\[ h_5 = \text{Union}(h_1, h_2) \]
Disjoint Sets (aka Union Find)

Disjoint Sets Data Structure

- **Handle MakeSet(Item x)**
  
  Precondition: none of the existing sets contains \( x \)
  
  Behaviour: create a new set \( \{x\} \) and return its handle

- **Handle FindSet(Item x)**
  
  Precondition: there exists a set that contains \( x \) (given pointer to \( x \))
  
  Behaviour: return the handle of the set that contains \( x \)

- **Handle Union(Handle h, Handle g)**
  
  Precondition: \( h \neq g \)
  
  Behaviour: merge two disjoint sets and return handle of new set

\[
\begin{align*}
h_5 &= \text{Union}(h_1, h_2) \\
h_3 &= h_4
\end{align*}
\]
Disjoint Sets (aka Union Find)

Disjoint Sets Data Structure

- **Handle MakeSet(Item x)**
  - Precondition: none of the existing sets contains x
  - Behaviour: create a new set \{x\} and return its handle

- **Handle FindSet(Item x)**
  - Precondition: there exists a set that contains x (given pointer to x)
  - Behaviour: return the handle of the set that contains x

- **Handle Union(Handle h, Handle g)**
  - Precondition: h \neq g
  - Behaviour: merge two **disjoint** sets and return handle of new set

\[h_5 = \text{Union}(h_1, h_2)\]
Disjoint Sets (aka Union Find)

Disjoint Sets Data Structure

- **Handle MakeSet(Item x)**
  Precondition: none of the existing sets contains x
  Behaviour: create a new set \{x\} and return its handle

- **Handle FindSet(Item x)**
  Precondition: there exists a set that contains x (given pointer to x)
  Behaviour: return the handle of the set that contains x

- **Handle Union(Handle h, Handle g)**
  Precondition: h ≠ g
  Behaviour: merge two disjoint sets and return handle of new set

\[ h_5 = \text{Union}(h_1, h_2) \]
First Attempt: List Implementation

Union \((h_1, h_2)\)

Need to find last element!

FindSet \((z_3)\)

Need to update all backward pointers!
First Attempt: List Implementation

**U**NION-Operation

**Union**($h_1, h_2$)

![Diagram of Union operation](image)
First Attempt: List Implementation

**Union(-Operation)**

Add extra pointer to the last element in each list

\[ \text{Union}(h_1, h_2) \]

Need to find last element!

FindSet \((z_3)\)

\[ h_4 \]

\[ x_1 \rightarrow x_2 \rightarrow x_3 \]

\[ y_1 \rightarrow y_2 \]

Need to update all backward pointers!
First Attempt: List Implementation

**UNION-Operation**

- Add extra pointer to the last element in each list

**Union**($h_1, h_2$)  

Need to find last element!
First Attempt: List Implementation

**UNION-Operation**

- Add extra pointer to the last element in each list

**Union** \((h_1, h_2)\)

```
Need to find last element!
```

```
\begin{align*}
& h_1 \\
& y_1 \\
& h_2 \\
& y_2 \\
\end{align*}
```

```
\begin{align*}
& x_1 \rightarrow x_2 \rightarrow x_3 \\
& x_1 \rightarrow x_2 \rightarrow x_3 \\
& h_1 \quad h_2 \\
\end{align*}
```

```
Need to update all backward pointers!
```
First Attempt: List Implementation

**UNION-Operation**

- Add extra pointer to the last element in each list

⇒ UNION takes constant time

\[
\text{Union}(h_1, h_2)
\]

- Need to find last element!

- Need to update all backward pointers!
**First Attempt: List Implementation**

**UNION-Operation**
- Add *extra pointer* to the last element in each list
  ⇒ UNION takes constant time

**FIND-Operation**

**Union** \((h_1, h_2)\)

\[
\begin{align*}
&X_1 \rightarrow X_2 \rightarrow X_3 \\
&Y_1 \rightarrow Y_2
\end{align*}
\]
**First Attempt: List Implementation**

**UNION-Operation**
- Add *extra pointer* to the last element in each list
  ⇒ UNION takes constant time

**FIND-Operation**

---

**Union** $(h_1, h_2)$

- $h_1$
- $h_2$

**FindSet** $(z_3)$

- $h_4$
### First Attempt: List Implementation

**UNION-Operation**
- Add extra pointer to the last element in each list
  ⇒ UNION takes constant time

**FIND-Operation**
- Add backward pointer to the list head from everywhere

**Union** $(h_1, h_2)$

```
    h1
   ↓   ↓
x1  x2  x3
     ↘     △
     y1  y2
```

Need to find last element!

**FindSet** $(z_3)$

```
    h4
   ↓   ↓
Z1  Z2  Z3  Z4
```

Need to update all backward pointers!
**First Attempt: List Implementation**

**UNION-Operation**
- Add extra pointer to the last element in each list
  ⇒ UNION takes constant time

**FIND-Operation**
- Add backward pointer to the list head from everywhere

**Union** \( (h_1, h_2) \)

**FindSet** \( (z_3) \)
First Attempt: List Implementation

**UNION-Operation**
- Add **extra pointer** to the last element in each list
  \[ \Rightarrow \text{UNION takes constant time} \]

**FIND-Operation**
- Add **backward pointer** to the list head from everywhere
  \[ \Rightarrow \text{FIND takes constant time} \]

**Union** \((h_1, h_2)\)

**FindSet** \((z_3)\)
First Attempt: List Implementation

**UNION-Operation**
- Add extra pointer to the last element in each list
  ⇒ UNION takes constant time

**FIND-Operation**
- Add backward pointer to the list head from everywhere
  ⇒ FIND takes constant time

- **Union**($h_1, h_2$)

- **FindSet**($z_3$)

---

5.3: Disjoint Sets
First Attempt: List Implementation

**UNION-Operation**
- Add extra pointer to the last element in each list
  ⇒ UNION takes constant time

**FIND-Operation**
- Add backward pointer to the list head from everywhere
  ⇒ FIND takes constant time

**Union**($h_1, h_2$)

Need to find last element!

**FindSet**($z_3$)

Need to update all backward pointers!
**First Attempt: List Implementation**

**UNION-Operation**
- Add extra pointer to the last element in each list
  \[ \Rightarrow \text{UNION takes constant time} \]

**FIND-Operation**
- Add backward pointer to the list head from everywhere
  \[ \Rightarrow \text{FIND takes constant time} \]

---

**Union** \((h_1, h_2)\)

- Need to find last element!

**FindSet** \((z_3)\)

- Need to update all backward pointers!
First Attempt: List Implementation (Analysis)

\[ d = \text{DisjointSet()} \]
First Attempt: List Implementation (Analysis)

\[ d = \text{DisjointSet()} \]
\[ h_0 = d.\text{MakeSet}(x_0) \]
First Attempt: List Implementation (Analysis)

\[ d = \text{DisjointSet}() \]
\[ h_0 = d.\text{MakeSet}(x_0) \]

\[ h_1 = d.\text{MakeSet}(x_1) \]

\[
\begin{array}{cc}
\text{Cost for } n \text{ UNION operations: } & \sum_{i=1}^{n} i = \Theta(n^2) \\
\text{better to append shorter list to longer} & \Rightarrow \text{Weighted-Union Heuristic}
\end{array}
\]
First Attempt: List Implementation (Analysis)

\[ d = \text{DisjointSet}() \]
\[ h_0 = d.\text{MakeSet}(x_0) \]

\[ h_1 = d.\text{MakeSet}(x_1) \]
\[ h_0 = d.\text{Union}(h_1, h_0) \]
First Attempt: List Implementation (Analysis)

\[ d = \text{DisjointSet()} \]
\[ h_0 = d.\text{MakeSet}(x_0) \]

\[ h_1 = d.\text{MakeSet}(x_1) \]
\[ h_0 = d.\text{Union}(h_1, h_0) \]

Cost for \( n \) \text{UNION} operations:

\[ \sum_{i=1}^{n} i = \Theta(n^2) \]

better to append shorter list to longer

\( \Rightarrow \)

Weighted-Union Heuristic
First Attempt: List Implementation (Analysis)

\[ d = \text{DisjointSet}() \]
\[ h_0 = d.\text{MakeSet}(x_0) \]

\[ h_1 = d.\text{MakeSet}(x_1) \]
\[ h_0 = d.\text{Union}(h_1, h_0) \]

Cost for \( n \) UNION operations:
\[ \sum_{i=1}^{n} i = \Theta(n^2) \]

better to append shorter list to longer

\[ \Rightarrow \]

Weighted-Union Heuristic

5.3: Disjoint Sets
First Attempt: List Implementation (Analysis)

\( d = \text{DisjointSet}() \)
\( h_0 = d.\text{MakeSet}(x_0) \)

\( h_1 = d.\text{MakeSet}(x_1) \)
\( h_0 = d.\text{Union}(h_1, h_0) \)

Cost for \( n \) UNION operations:

\[ \sum_{i=1}^{n} i = \Theta(n^2) \]
First Attempt: List Implementation (Analysis)

\[ d = \text{DisjointSet}(\) \]
\[ h_0 = d.\text{MakeSet}(x_0) \]

\[ h_1 = d.\text{MakeSet}(x_1) \]
\[ h_0 = d.\text{Union}(h_1, h_0) \]

Better to append shorter list to longer ⇝ Weighted-Union Heuristic
First Attempt: List Implementation (Analysis)

\( d = \text{DisjointSet}() \)
\( h_0 = d.\text{MakeSet}(x_0) \)

\( h_1 = d.\text{MakeSet}(x_1) \)
\( h_0 = d.\text{Union}(h_1, h_0) \)
\( h_2 = d.\text{MakeSet}(x_2) \)

\[ \sum_{i=1}^{n} \text{Cost for n UNION operations:} \]
\[ \Theta(n^2) \]

Better to append shorter list to longer \( \Rightarrow \) Weighted-Union Heuristic
First Attempt: List Implementation (Analysis)

\[ d = \text{DisjointSet}() \]
\[ h_0 = d.\text{MakeSet}(x_0) \]

\[ h_1 = d.\text{MakeSet}(x_1) \]
\[ h_0 = d.\text{Union}(h_1, h_0) \]
\[ h_2 = d.\text{MakeSet}(x_2) \]
\[ h_0 = d.\text{Union}(h_2, h_0) \]
First Attempt: List Implementation (Analysis)

\[ d = \text{DisjointSet}() \]
\[ h_0 = d.\text{MakeSet}(x_0) \]

\[ h_1 = d.\text{MakeSet}(x_1) \]
\[ h_0 = d.\text{Union}(h_1, h_0) \]
\[ h_2 = d.\text{MakeSet}(x_2) \]
\[ h_0 = d.\text{Union}(h_2, h_0) \]
First Attempt: List Implementation (Analysis)

\[ \begin{align*}
    d &= \text{DisjointSet()} \\
    h_0 &= d.\text{MakeSet}(x_0) \\
    h_1 &= d.\text{MakeSet}(x_1) \\
    h_0 &= d.\text{Union}(h_1, h_0) \\
    h_2 &= d.\text{MakeSet}(x_2) \\
    h_0 &= d.\text{Union}(h_2, h_0) \\
\end{align*} \]

Cost for \( n \) UNION operations:
\[ \sum_{i=1}^{n} i = \Theta(n^2) \]

Better to append shorter list to longer → Weighted-Union Heuristic

5.3: Disjoint Sets
First Attempt: List Implementation (Analysis)

\[ d = \text{DisjointSet()} \]
\[ h_0 = d\.\text{MakeSet}(x_0) \]

\[ h_1 = d\.\text{MakeSet}(x_1) \]
\[ h_0 = d\.\text{Union}(h_1, h_0) \]
\[ h_2 = d\.\text{MakeSet}(x_2) \]
\[ h_0 = d\.\text{Union}(h_2, h_0) \]

Cost for \( n \) UNION operations:
\[ \sum_{i=1}^{n} i = \Theta(n^2) \]
First Attempt: List Implementation (Analysis)

\[ d = \text{DisjointSet()} \]
\[ h_0 = d.\text{MakeSet}(x_0) \]

\[ h_1 = d.\text{MakeSet}(x_1) \]
\[ h_0 = d.\text{Union}(h_1, h_0) \]
\[ h_2 = d.\text{MakeSet}(x_2) \]
\[ h_0 = d.\text{Union}(h_2, h_0) \]
First Attempt: List Implementation (Analysis)

\[ d = \text{DisjointSet}() \]
\[ h_0 = d.\text{MakeSet}(x_0) \]

\[ h_1 = d.\text{MakeSet}(x_1) \]
\[ h_0 = d.\text{Union}(h_1, h_0) \]
\[ h_2 = d.\text{MakeSet}(x_2) \]
\[ h_0 = d.\text{Union}(h_2, h_0) \]
\[ h_3 = d.\text{MakeSet}(x_3) \]
First Attempt: List Implementation (Analysis)

\[
d = \text{DisjointSet()}
\]
\[
h_0 = d.\text{MakeSet}(x_0)
\]

\[
h_1 = d.\text{MakeSet}(x_1)
\]
\[
h_0 = d.\text{Union}(h_1, h_0)
\]
\[
h_2 = d.\text{MakeSet}(x_2)
\]
\[
h_0 = d.\text{Union}(h_2, h_0)
\]
\[
h_3 = d.\text{MakeSet}(x_3)
\]
\[
h_0 = d.\text{Union}(h_3, h_0)
\]
First Attempt: List Implementation (Analysis)

\[ d = \text{DisjointSet()} \]
\[ h_0 = d.\text{MakeSet}(x_0) \]

\[ h_1 = d.\text{MakeSet}(x_1) \]
\[ h_0 = d.\text{Union}(h_1, h_0) \]
\[ h_2 = d.\text{MakeSet}(x_2) \]
\[ h_0 = d.\text{Union}(h_2, h_0) \]
\[ h_3 = d.\text{MakeSet}(x_3) \]
\[ h_0 = d.\text{Union}(h_3, h_0) \]
First Attempt: List Implementation (Analysis)

\[ d = \text{DisjointSet()} \]
\[ h_0 = d.\text{MakeSet}(x_0) \]

\[ h_1 = d.\text{MakeSet}(x_1) \]
\[ h_0 = d.\text{Union}(h_1, h_0) \]
\[ h_2 = d.\text{MakeSet}(x_2) \]
\[ h_0 = d.\text{Union}(h_2, h_0) \]
\[ h_3 = d.\text{MakeSet}(x_3) \]
\[ h_0 = d.\text{Union}(h_3, h_0) \]
First Attempt: List Implementation (Analysis)

\[ d = \text{DisjointSet}() \]
\[ h_0 = d.\text{MakeSet}(x_0) \]

\[ h_1 = d.\text{MakeSet}(x_1) \]
\[ h_0 = d.\text{Union}(h_1, h_0) \]

\[ h_2 = d.\text{MakeSet}(x_2) \]
\[ h_0 = d.\text{Union}(h_2, h_0) \]

\[ h_3 = d.\text{MakeSet}(x_3) \]
\[ h_0 = d.\text{Union}(h_3, h_0) \]
First Attempt: List Implementation (Analysis)

d = DisjointSet()
\( h_0 = d.\text{MakeSet}(x_0) \)

\( h_1 = d.\text{MakeSet}(x_1) \)
\( h_0 = d.\text{Union}(h_1, h_0) \)
\( h_2 = d.\text{MakeSet}(x_2) \)
\( h_0 = d.\text{Union}(h_2, h_0) \)
\( h_3 = d.\text{MakeSet}(x_3) \)
\( h_0 = d.\text{Union}(h_3, h_0) \)

Cost for \( n \text{ UNION operations:} \)
\[ \sum_{i=1}^{n} i = \Theta(n^2) \]

better to append shorter list to longer 
\( \Rightarrow \) Weighted-Union Heuristic
First Attempt: List Implementation (Analysis)

\[ d = \text{DisjointSet()} \]
\[ h_0 = d.\text{MakeSet}(x_0) \]

\[ h_1 = d.\text{MakeSet}(x_1) \]
\[ h_0 = d.\text{Union}(h_1, h_0) \]
\[ h_2 = d.\text{MakeSet}(x_2) \]
\[ h_0 = d.\text{Union}(h_2, h_0) \]
\[ h_3 = d.\text{MakeSet}(x_3) \]
\[ h_0 = d.\text{Union}(h_3, h_0) \]

Cost for \( n \) UNION operations: \( \sum_{i=1}^{n} i = \Theta(n^2) \)
First Attempt: List Implementation (Analysis)

\[ d = \text{DisjointSet}() \]
\[ h_0 = d.\text{MakeSet}(x_0) \]

\[ h_1 = d.\text{MakeSet}(x_1) \]
\[ h_0 = d.\text{Union}(h_1, h_0) \]
\[ h_2 = d.\text{MakeSet}(x_2) \]
\[ h_0 = d.\text{Union}(h_2, h_0) \]
\[ h_3 = d.\text{MakeSet}(x_3) \]
\[ h_0 = d.\text{Union}(h_3, h_0) \]

Cost for \( n \) UNION operations: \( \sum_{i=1}^{n} i = \Theta(n^2) \)

better to append shorter list to longer \( \rightsquigarrow \) Weighted-Union Heuristic
Weighted-Union Heuristic

- Keep track of the length of each list
Weighted-Union Heuristic

- Keep track of the length of each list
- Append shorter list to the longer list (breaking ties arbitrarily)
Weighted-Union Heuristic

- Keep track of the length of each list
- Append shorter list to the longer list (breaking ties arbitrarily)

can be done easily without significant overhead

Theorem 21.1: Amortized Analysis
Every operation has amortized cost $O(\log n)$, but there may be operations with total cost $\Theta(n)$. 

5.3: Disjoint Sets
Weighted-Union Heuristic

- Keep track of the length of each list
- Append shorter list to the longer list (breaking ties arbitrarily)

can be done easily without significant overhead

Theorem 21.1

Using the Weighted-Union heuristic, any sequence of $m$ operations, $n$ of which are MAKE-SET operations, takes $O(m + n \cdot \log n)$ time.
Weighted-Union Heuristic

- Keep track of the length of each list
- Append shorter list to the longer list (breaking ties arbitrarily)

The weighted-union heuristic can be done easily without significant overhead.

Theorem 21.1

Using the Weighted-Union heuristic, any sequence of $m$ operations, $n$ of which are MAKE-SET operations, takes $O(m + n \cdot \log n)$ time.

Amortized Analysis: Every operation has amortized cost $O(\log n)$, but there may be operations with total cost $\Theta(n)$. 
Analysis of Weighted-Union Heuristic

Using the **Weighted-Union heuristic**, any sequence of $m$ operations, $n$ of which are MAKE-SET operations, takes $O(m + n \cdot \log n)$ time.

**Theorem 21.1**
Using the Weighted-Union heuristic, any sequence of $m$ operations, $n$ of which are MAKE-SET operations, takes $\mathcal{O}(m + n \cdot \log n)$ time.

Proof:
Using the Weighted-Union heuristic, any sequence of \( m \) operations, \( n \) of which are MAKE-SET operations, takes \( O(m + n \cdot \log n) \) time.

Proof:

- \( n \) MAKE-SET operations \( \Rightarrow \) at most \( n - 1 \) UNION operations
Analysis of Weighted-Union Heuristic

Theorem 21.1

Using the **Weighted-Union heuristic**, any sequence of $m$ operations, $n$ of which are MAKE-SET operations, takes $O(m + n \cdot \log n)$ time.

Proof:

- $n$ MAKE-SET operations $\Rightarrow$ at most $n - 1$ UNION operations
- Consider element $x$ and the number of updates of the backward pointer
Using the Weighted-Union heuristic, any sequence of \( m \) operations, \( n \) of which are MAKE-SET operations, takes \( \mathcal{O}(m + n \cdot \log n) \) time.

**Proof:**

- \( n \) MAKE-SET operations \( \Rightarrow \) at most \( n - 1 \) UNION operations
- Consider element \( x \) and the number of updates of the backward pointer
Using the Weighted-Union heuristic, any sequence of \( m \) operations, \( n \) of which are MAKE-SET operations, takes \( \mathcal{O}(m + n \cdot \log n) \) time.

**Proof:**

- \( n \) MAKE-SET operations \( \Rightarrow \) at most \( n - 1 \) UNION operations
- Consider element \( x \) and the number of updates of the backward pointer
Analysis of Weighted-Union Heuristic

Using the Weighted-Union heuristic, any sequence of \( m \) operations, \( n \) of which are MAKE-SET operations, takes \( \mathcal{O}(m + n \cdot \log n) \) time.

Proof:
- \( n \) MAKE-SET operations \( \Rightarrow \) at most \( n - 1 \) UNION operations
- Consider element \( x \) and the number of updates of the backward pointer
Analysis of Weighted-Union Heuristic

Using the Weighted-Union heuristic, any sequence of $m$ operations, $n$ of which are MAKE-SET operations, takes $O(m + n \cdot \log n)$ time.

Proof:
- $n$ MAKE-SET operations $\Rightarrow$ at most $n - 1$ UNION operations
- Consider element $x$ and the number of updates of the backward pointer
- After each update of $x$, its set increases by a factor of at least 2
Using the Weighted-Union heuristic, any sequence of \( m \) operations, \( n \) of which are MAKE-SET operations, takes \( \mathcal{O}(m + n \cdot \log n) \) time.

Proof:

- \( n \) MAKE-SET operations \( \Rightarrow \) at most \( n - 1 \) UNION operations
- Consider element \( x \) and the number of updates of the backward pointer
- After each update of \( x \), its set increases by a factor of at least 2
- \( \Rightarrow \) Backward pointer of \( x \) is updated at most \( \log_2 n \) times
Using the **Weighted-Union heuristic**, any sequence of $m$ operations, $n$ of which are MAKE-SET operations, takes $O(m + n \cdot \log n)$ time.

**Proof:**

- $n$ MAKE-SET operations $\Rightarrow$ at most $n - 1$ UNION operations
- Consider element $x$ and the number of updates of the backward pointer
- After each update of $x$, its set increases by a factor of at least 2
  $\Rightarrow$ Backward pointer of $x$ is updated at most $\log_2 n$ times
- Other updates for UNION, MAKE-SET & FIND-SET take $O(1)$ time per operation
Analysis of Weighted-Union Heuristic

Using the Weighted-Union heuristic, any sequence of $m$ operations, $n$ of which are MAKE-SET operations, takes $O(m + n \cdot \log n)$ time.

**Theorem 21.1**

Proof:

- $n$ MAKE-SET operations $\Rightarrow$ at most $n - 1$ UNION operations
- Consider element $x$ and the number of updates of the backward pointer
- After each update of $x$, its set increases by a factor of at least 2
  $\Rightarrow$ Backward pointer of $x$ is updated at most $\log_2 n$ times
- Other updates for UNION, MAKE-SET & FIND-SET take $O(1)$ time per operation
Using the Weighted-Union heuristic, any sequence of \( m \) operations, \( n \) of which are MAKE-SET operations, takes \( \mathcal{O}(m + n \cdot \log n) \) time.

**Proof:**

- \( n \) MAKE-SET operations \( \Rightarrow \) at most \( n - 1 \) UNION operations
- Consider element \( x \) and the number of updates of the backward pointer
- After each update of \( x \), its set increases by a factor of at least 2

\( \Rightarrow \) Backward pointer of \( x \) is updated at most \( \log_2 n \) times

- Other updates for UNION, MAKE-SET & FIND-SET take \( \mathcal{O}(1) \) time per operation
How to Improve?

Basic Idea: Update Backward

- **MAKE-SET:** $O(1)$
- **FIND-SET:** $O(n)$
- **UNION:** $O(1)$ (amortized)

Doubly-Linked List

- **MAKE-SET:** $O(1)$
- **FIND-SET:** $O(n)$
- **UNION:** $O(1)$
How to Improve?

**Doubly-Linked List**
- **MAKE-SET**: $O(1)$
- **FIND-SET**: $O(n)$
- **UNION**: $O(1)$

**Weighted-Union Heuristic**
- **MAKE-SET**: $O(1)$
- **FIND-SET**: $O(1)$
- **UNION**: $O(\log n)$ (amortized)
How to Improve?

Basic Idea: Update Backward Pointers only during FIND

Doubly-Linked List
- MAKE-SET: $O(1)$
- FIND-SET: $O(n)$
- UNION: $O(1)$

Weighted-Union Heuristic
- MAKE-SET: $O(1)$
- FIND-SET: $O(1)$
- UNION: $O(\log n)$ (amortized)
Disjoint Sets via Forests

Forest Structure

- Set is represented by a **rooted tree** with root being the representative.
- Every node has pointer .\( p \) to its parent (for root \( x \), \( x.p = x \))
Disjoint Sets via Forests

\( \{b, c, e, h\} \)

### Forest Structure
- Set is represented by a **rooted tree** with root being the representative.
- Every node has **pointer** \( .p \) to its parent (for root \( x \), \( x.p = x \))
Disjoint Sets via Forests

\{b, c, e, h\}

**Forest Structure**
- Set is represented by a rooted tree with root being the representative
- Every node has pointer $p$ to its parent (for root $x$, $x.p = x$)
- UNION: Merge the two trees
Disjoint Sets via Forests

\{b, c, e, h\} \quad \{d, f, g\}

Forest Structure

- Set is represented by a rooted tree with root being the representative
- Every node has pointer \( p \) to its parent (for root \( x \), \( x.p = x \))
- UNION: Merge the two trees
Disjoint Sets via Forests

Set is represented by a rooted tree with root being the representative.
Every node has pointer $p$ to its parent (for root $x$, $x.p = x$).
UNION: Merge the two trees.
Disjoint Sets via Forests

\{b, c, e, h\} \quad \{d, f, g\} \quad \{b, c, d, e, f, g, h\}

Forest Structure

- Set is represented by a rooted tree with root being the representative
- Every node has pointer \(p\) to its parent (for root \(x\), \(x.p = x\))
- UNION: Merge the two trees

Append tree of smaller height \(\leadsto\) Union by Rank
Disjoint Sets via Forests

\{ b, c, e, h \} \quad \{ d, f, g \} \quad \{ b, c, d, e, f, g, h \}

Forest Structure

- Set is represented by a rooted tree with root being the representative
- Every node has pointer \( p \) to its parent (for root \( x \), \( x.p = x \))
- UNION: Merge the two trees

**Append tree of smaller height \( \rightsquigarrow \) Union by Rank**
Disjoint Sets via Forests

\{b, c, e, h\} \quad \{d, f, g\} \quad \{b, c, d, e, f, g, h\}

Rank may be just an upper bound on the height!

Forest Structure

- Set is represented by a rooted tree with root being the representative
- Every node has pointer \(p\) to its parent (for root \(x, x.p = x\))
- UNION: Merge the two trees

Append tree of smaller height \(\leadsto\) Union by Rank
Disjoint Sets via Forests

\[ \{b, c, e, h\} \quad \{d, f, g\} \quad \{b, c, d, e, f, g, h\} \]

Rank may be just an upper bound on the height!

Forest Structure

- Set is represented by a rooted tree with root being the representative
- Every node has pointer \( p \) to its parent (for root \( x \), \( x.p = x \))
- UNION: Merge the two trees

Append tree of smaller height \( \sim \) Union by Rank
Path Compression during FIND-SET

0: \textbf{FindSet}(x)
1: \textbf{if} \ x \neq x.p
2: \> x.p = \textbf{FindSet}(x.p)
3: \textbf{return} \ x.p

Diagram:

- Node f
- Node c
- Nodes d and g
- Nodes h and e
- Node b
Path Compression during FIND-SET

0: \textbf{FindSet} (x)
1: \textbf{if} \ x \neq x.p
2: \hspace{1em} x.p = \textbf{FindSet} (x.p)
3: \textbf{return} \ x.p
Path Compression during FIND-SET

0: FindSet (x)
1: if x \neq x.p
2: x.p = FindSet (x.p)
3: return x.p
Path Compression during FIND-SET

0: \textbf{FindSet}(x)
1: \textbf{if} $x \neq x.p$
2: \hspace{1em} $x.p = \textbf{FindSet}(x.p)$
3: \textbf{return} $x.p$

Maintaining the exact height would be costly, hence rank is only an upper bound!
Path Compression during FIND-SET

0: \textbf{FindSet}(x)
1: \hspace{1em} \textbf{if} \ x \neq x.p
2: \hspace{2em} x.p = \textbf{FindSet}(x.p)
3: \hspace{1em} \textbf{return} \ x.p
Path Compression during FIND-SET

0: \textbf{FindSet} (x)
1: \textbf{if } x \neq x.p
2: \hspace{1em} x.p = \textbf{FindSet} (x.p)
3: \textbf{return } x.p
Path Compression during FIND-SET

0: FindSet(x)
1: if x \neq x.p
2: x.p = FindSet(x.p)
3: return x.p
Path Compression during FIND-SET

```
0: FindSet(x)
1: if x ≠ x.p
2: x.p = FindSet(x.p)
3: return x.p
```
Path Compression during FIND-SET

0: \textbf{FindSet}(x)
1: \textbf{if } x \neq x.p
2: \quad x.p = \textbf{FindSet}(x.p)
3: \quad \textbf{return } x.p
Path Compression during FIND-SET

0: FindSet(\(x\))
1: \textbf{if} \(x \neq x.p\)
2: \(x.p = \text{FindSet}(x.p)\)
3: \textbf{return} \(x.p\)
Path Compression during FIND-SET

0: **FindSet** (x)
1: \[ \textbf{if} \ x \neq x.p \]
2: \[ x.p = \textbf{FindSet} (x.p) \]
3: \[ \textbf{return} \ x.p \]
Path Compression during FIND-SET

0: \textbf{FindSet}(x)
1: \textbf{if} \ x \neq x.p
2: \ x.p = \textbf{FindSet}(x.p)
3: \textbf{return} \ x.p
Path Compression during FIND-SET

0: \textbf{FindSet}(x)
1: \textbf{if} \ x \neq x.p
2: \hspace{1em} x.p = \textbf{FindSet}(x.p)
3: \textbf{return} \ x.p
Path Compression during FIND-SET

0: \textbf{FindSet}(x)
1: \textbf{if } x \neq x.p
2: \hspace{1em} x.p = \textbf{FindSet}(x.p)
3: \hspace{1em} \textbf{return } x.p
Path Compression during FIND-SET

0: FindSet(x)
1: if x ≠ x.p
2: x.p = FindSet(x.p)
3: return x.p
Path Compression during FIND-SET

0: \textbf{FindSet} (x)
1: \textbf{if} \hspace{1em} x \neq x.p
2: \hspace{1em} x.p = \textbf{FindSet} (x.p)
3: \textbf{return} \hspace{1em} x.p
0: **FindSet**(*x*)
1: \textbf{if } *x* \neq *x*.*p*
2: \hspace{1em} *x*.*p* = **FindSet**(*x*.*p*)
3: \textbf{return } *x*.*p*
Path Compression during FIND-SET

0: \textbf{FindSet}(x)
1: \hspace{1em} \textbf{if} \ x \neq x.p
2: \hspace{4em} x.p = \textbf{FindSet}(x.p)
3: \hspace{4em} \textbf{return} \ x.p
Path Compression during FIND-SET

0: \textbf{FindSet}(x)
1: \textbf{if} \ x \neq x.p
2: \ x.p = \textbf{FindSet}(x.p)
3: \textbf{return} \ x.p
Path Compression during FIND-SET

Maintaining the exact height would be costly, hence rank is only an **upper bound**!

```
0: FindSet(x)
1: if x ≠ x.p
2: x.p = FindSet(x.p)
3: return x.p
```
Combining Union by Rank and Path Compression

**Theorem 21.14**

Any sequence of $m$ MAKE-SET, UNION, FIND-SET operations, $n$ of which are MAKE-SET operations, can be performed in $O(m \cdot \alpha(n))$ time.
Combining Union by Rank and Path Compression

**Theorem 21.14**

Any sequence of $m$ MAKE-SET, UNION, FIND-SET operations, $n$ of which are MAKE-SET operations, can be performed in $O(m \cdot \alpha(n))$ time.

$$
\alpha(n) = \begin{cases} 
0 & \text{for } 0 \leq n \leq 2, \\
1 & \text{for } n = 3, \\
2 & \text{for } 4 \leq n \leq 7, \\
3 & \text{for } 8 \leq n \leq 2047, \\
4 & \text{for } 2048 \leq n \leq 10^{80}
\end{cases}
$$
Combining Union by Rank and Path Compression

Theorem 21.14

Any sequence of $m$ MAKE-SET, UNION, FIND-SET operations, $n$ of which are MAKE-SET operations, can be performed in $O(m \cdot \alpha(n))$ time.

$$
\alpha(n) = \begin{cases} 
0 & \text{for } 0 \leq n \leq 2, \\
1 & \text{for } n = 3, \\
2 & \text{for } 4 \leq n \leq 7, \\
3 & \text{for } 8 \leq n \leq 2047, \\
4 & \text{for } 2048 \leq n \leq 10^{80} 
\end{cases}
$$

More than the number of atoms in the universe!
**Theorem 21.14**

Any sequence of $m$ MAKE-SET, UNION, FIND-SET operations, $n$ of which are MAKE-SET operations, can be performed in $O(m \cdot \alpha(n))$ time.

\[ \alpha(n) = \begin{cases} 
0 & \text{for } 0 \leq n \leq 2, \\
1 & \text{for } n = 3, \\
2 & \text{for } 4 \leq n \leq 7, \\
3 & \text{for } 8 \leq n \leq 2047, \\
4 & \text{for } 2048 \leq n \leq 10^{80} 
\end{cases} \]

\(\log^*(n)\), the iterated logarithm, satisfies \(\alpha(n) \leq \log^*(n)\), but still \(\log^*(10^{80}) = 5\).
Combining Union by Rank and Path Compression

Theorem 21.14
Any sequence of \( m \) MAKE-SET, UNION, FIND-SET operations, \( n \) of which are MAKE-SET operations, can be performed in \( O(m \cdot \alpha(n)) \) time.

In practice, \( \alpha(n) \) is a small constant

\[
\alpha(n) = \begin{cases} 
0 & \text{for } 0 \leq n \leq 2, \\
1 & \text{for } n = 3, \\
2 & \text{for } 4 \leq n \leq 7, \\
3 & \text{for } 8 \leq n \leq 2047, \\
4 & \text{for } 2048 \leq n \leq 10^{80} 
\end{cases}
\]
Combining Union by Rank and Path Compression

Data Structure is essentially optimal! (for more details see CLRS)

**Theorem 21.14**

Any sequence of \( m \) MAKE-SET, UNION, FIND-SET operations, \( n \) of which are MAKE-SET operations, can be performed in \( O(m \cdot \alpha(n)) \) time.

In practice, \( \alpha(n) \) is a small constant

\[
\alpha(n) = \begin{cases} 
0 & \text{for } 0 \leq n \leq 2, \\
1 & \text{for } n = 3, \\
2 & \text{for } 4 \leq n \leq 7, \\
3 & \text{for } 8 \leq n \leq 2047, \\
4 & \text{for } 2048 \leq n \leq 10^{80}
\end{cases}
\]
Simulating the Effects of Union by Rank and Path Compression

1. Initialise singletons 1, 2, ..., 300

2. For every $1 \leq i \leq 300$, pick a random $1 \leq r \leq 300$, $r \neq i$ and perform $UNION(\text{FIND}(i), \text{FIND}(r))$

3. Perform $j \in \{0, 100, 200, 300, 600, 900\}$ many $\text{FIND}(r)$, where $1 \leq r \leq 300$ is random
Simulating the Effects of Union by Rank and Path Compression

Experimental Setup

1. Initialise singletons 1, 2, \ldots, 300
2. For every $1 \leq i \leq 300$, pick a random $1 \leq r \leq 300$, $r \neq i$ and perform \textsc{Union}(\textsc{Find}(i), \textsc{Find}(r))
3. Perform $j \in \{0, 100, 200, 300, 600, 900\}$ many \textsc{Find}(r), where $1 \leq r \leq 300$ is random
Union by Rank without Path Compression
Union by Rank with Path Compression
After 100 additional FINDs
After 200 additional FINDs
After 300 additional FINDs
After 600 additional FINDs
After 900 additional FINDs
Outline

Disjoint Sets

Introduction to Graphs and Graph Searching
Origin of Graph Theory

Seven Bridges at Königsberg 1737

Origin of Graph Theory

Seven Bridges at Königsberg 1737

Leonhard Euler (1707-1783)

Is there a tour which crosses each bridge exactly once?
Origin of Graph Theory

Leonhard Euler (1707-1783)

Seven Bridges at Königsberg 1737

Is there a tour which crosses each bridge exactly once?

Leonhard Euler (1707-1783)


Is there a tour which visits every island exactly once?

Origin of Graph Theory

Seven Bridges at Königsberg 1737

Is there a tour which crosses each bridge exactly once?

Leonhard Euler (1707-1783)


Is there a tour which visits every island exactly once?
Origin of Graph Theory

Leonhard Euler (1707-1783)

Seven Bridges at Königsberg 1737

Is there a tour which crosses each bridge exactly once?

Is there a tour which visits every island exactly once?


5.3: Disjoint Sets
Origin of Graph Theory

Seven Bridges at Königsberg 1737

Is there a tour which crosses each bridge exactly once?

Is there a tour which visits every island exactly once?

Leonhard Euler (1707-1783)


5.3: Disjoint Sets
Origin of Graph Theory

Seven Bridges at Königsberg 1737

Is there a tour which crosses each bridge exactly once?

Is there a tour which visits every island exactly once?

⇝ 1B course: Complexity Theory

Leonhard Euler (1707-1783)


5.3: Disjoint Sets
Direct Graph

A graph $G = (V, E)$ consists of:

- $V$: the set of vertices
- $E$: the set of edges (arcs)
What is a Graph?

**Directed Graph**

A graph $G = (V, E)$ consists of:
- $V$: the set of vertices
- $E$: the set of edges (arcs)
What is a Graph?

A graph \( G = (V, E) \) consists of:
- \( V \): the set of vertices
- \( E \): the set of edges (arcs)

Directed Graph

\[
\begin{align*}
V &= \{1, 2, 3, 4\} \\
E &= \{(1, 2), (1, 3), (2, 3), (3, 1), (3, 4)\}
\end{align*}
\]
What is a Graph?

Directed Graph
A graph $G = (V, E)$ consists of:
- $V$: the set of vertices
- $E$: the set of edges (arcs)

Undirected Graph
A graph $G = (V, E)$ consists of:
- $V$: the set of vertices
- $E$: the set of (undirected) edges

$V = \{1, 2, 3, 4\}$
$E = \{(1, 2), (1, 3), (2, 3), (3, 1), (3, 4)\}$
**What is a Graph?**

### Directed Graph
A graph $G = (V, E)$ consists of:
- $V$: the set of vertices
- $E$: the set of edges (arcs)

### Undirected Graph
A graph $G = (V, E)$ consists of:
- $V$: the set of vertices
- $E$: the set of (undirected) edges

**Example Directed Graph**
- $V = \{1, 2, 3, 4\}$
- $E = \{(1, 2), (1, 3), (2, 3), (3, 1), (3, 4)\}$

**Example Undirected Graph**
- $V = \{1, 2, 3, 4\}$
- $E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{3, 4\}\}$
What is a Graph?

**Directed Graph**
A graph $G = (V, E)$ consists of:
- $V$: the set of vertices
- $E$: the set of edges (arcs)

**Undirected Graph**
A graph $G = (V, E)$ consists of:
- $V$: the set of vertices
- $E$: the set of (undirected) edges

**Paths and Connectivity**
A sequence of edges between two vertices forms a path

---

**Example**: For the graph $G = (V, E)$ with $V = \{1, 2, 3, 4\}$ and $E = \{(1, 2), (1, 3), (2, 3), (3, 1), (3, 4)\}$, the path $p = (1, 2, 3, 4)$ is a cycle.

---

**Example**: For the graph $G = (V, E)$ with $V = \{1, 2, 3, 4\}$ and $E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{3, 4\}\}$, $G$ is connected.
What is a Graph?

Directed Graph

A graph $G = (V, E)$ consists of:
- $V$: the set of vertices
- $E$: the set of edges (arcs)

Undirected Graph

A graph $G = (V, E)$ consists of:
- $V$: the set of vertices
- $E$: the set of (undirected) edges

Paths and Connectivity

- A sequence of edges between two vertices forms a path

Path $p = (1, 2, 3, 4)

$V = \{1, 2, 3, 4\}$

$E = \{(1, 2), (1, 3), (2, 3), (3, 1), (3, 4)\}$
What is a Graph?

Directed Graph
A graph $G = (V, E)$ consists of:
- $V$: the set of vertices
- $E$: the set of edges (arcs)

Undirected Graph
A graph $G = (V, E)$ consists of:
- $V$: the set of vertices
- $E$: the set of (undirected) edges

Paths and Connectivity
- A sequence of edges between two vertices forms a path

Path $p = (1, 2, 3, 1)$, which is a cycle

$V = \{1, 2, 3, 4\}$
$E = \{(1, 2), (1, 3), (2, 3), (3, 1), (3, 4)\}$
What is a Graph?

Directed Graph
A graph \( G = (V, E) \) consists of:
- \( V \): the set of vertices
- \( E \): the set of edges (arcs)

Undirected Graph
A graph \( G = (V, E) \) consists of:
- \( V \): the set of vertices
- \( E \): the set of (undirected) edges

Paths and Connectivity
- A sequence of edges between two vertices forms a path

Directed Graph Example:
- \( V = \{1, 2, 3, 4\} \)
- \( E = \{(1, 2), (1, 3), (2, 3), (3, 1), (3, 4)\} \)

Undirected Graph Example:
- \( V = \{1, 2, 3, 4\} \)
- \( E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{3, 4\}\} \)
What is a Graph?

**Directed Graph**
A graph $G = (V, E)$ consists of:
- $V$: the set of vertices
- $E$: the set of edges (arcs)

**Undirected Graph**
A graph $G = (V, E)$ consists of:
- $V$: the set of vertices
- $E$: the set of (undirected) edges

**Paths and Connectivity**
- A sequence of edges between two vertices forms a path

**Example 1**
- $V = \{1, 2, 3, 4\}
- E = \{(1, 2), (1, 3), (2, 3), (3, 1), (3, 4)\}

**Example 2**
- $V = \{1, 2, 3, 4\}
- E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{3, 4\}\}

$G$ is not a DAG
What is a Graph?

Directed Graph
A graph $G = (V, E)$ consists of:
- $V$: the set of vertices
- $E$: the set of edges (arcs)

Undirected Graph
A graph $G = (V, E)$ consists of:
- $V$: the set of vertices
- $E$: the set of (undirected) edges

Paths and Connectivity
- A sequence of edges between two vertices forms a path
- If each pair of vertices has a path linking them, then $G$ is connected

$V = \{1, 2, 3, 4\}$
$E = \{(1, 2), (1, 3), (2, 3), (3, 1), (3, 4)\}$

$V = \{1, 2, 3, 4\}$
$E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{3, 4\}\}$
What is a Graph?

**Directed Graph**
A graph \( G = (V, E) \) consists of:
- \( V \): the set of vertices
- \( E \): the set of edges (arcs)

**Undirected Graph**
A graph \( G = (V, E) \) consists of:
- \( V \): the set of vertices
- \( E \): the set of (undirected) edges

**Paths and Connectivity**
- A sequence of edges between two vertices forms a path
- If each pair of vertices has a path linking them, then \( G \) is connected

**G is not a DAG**

**G is not (strongly) connected**

**Later: edge-weighted graphs**

\[ G = (V, E, w) \]

5.3: Disjoint Sets T.S. 22
What is a Graph?

A graph $G = (V, E)$ consists of:
- $V$: the set of vertices
- $E$: the set of edges (arcs)

Directed Graph

A graph $G = (V, E)$ consists of:
- $V$: the set of vertices
- $E$: the set of (undirected) edges

Undirected Graph

A graph $G = (V, E)$ consists of:
- $V$: the set of vertices
- $E$: the set of (undirected) edges

Later: edge-weighted graphs $G = (V, E, w)$

Paths and Connectivity

- A sequence of edges between two vertices forms a path
- If each pair of vertices has a path linking them, then $G$ is connected

$V = \{1, 2, 3, 4\}$
$E = \{(1, 2), (1, 3), (2, 3), (3, 1), (3, 4)\}$

$V = \{1, 2, 3, 4\}$
$E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{3, 4\}\}$

$G$ is not a DAG

$G$ is not (strongly) connected

$G$ is connected
Representations of Directed and Undirected Graphs

Figure 22.1 Two representations of an undirected graph. (a) An undirected graph $G$ with 5 vertices and 7 edges. (b) An adjacency-list representation of $G$. (c) The adjacency-matrix representation of $G$. 
Representations of Directed and Undirected Graphs

**Figure 22.1** Two representations of an undirected graph. (a) An undirected graph $G$ with 5 vertices and 7 edges. (b) An adjacency-list representation of $G$. (c) The adjacency-matrix representation of $G$.

**Figure 22.2** Two representations of a directed graph. (a) A directed graph $G$ with 6 vertices and 8 edges. (b) An adjacency-list representation of $G$. (c) The adjacency-matrix representation of $G$. 

shortest-paths algorithms presented in Chapter 25 assume that their input graphs are represented by adjacency matrices.

The adjacency-list representation of a graph $G = (V, E)$ consists of an array $\text{Adj}$ of $|V|$ lists, one for each vertex in $V$. For each $u \in V$, the adjacency list $\text{Adj}[u]$ consists of all the vertices adjacent to $u$ in $G$. (Alternatively, it may contain pointers to these vertices.) Since the adjacency lists represent the edges of a graph, in pseudocode we treat the array $\text{Adj}$ as an attribute of the graph, just as we treat the edge set $E$. In pseudocode, therefore, we will notation such as $G:\text{Adj}[u]$.
Representations of Directed and Undirected Graphs

Figure 22.1 Two representations of an undirected graph. (a) An undirected graph $G$ with 5 vertices and 7 edges. (b) An adjacency-list representation of $G$. (c) The adjacency-matrix representation of $G$.

Figure 22.2 Two representations of a directed graph. (a) A directed graph $G$ with 6 vertices and 8 edges. (b) An adjacency-list representation of $G$. (c) The adjacency-matrix representation of $G$. 

5.3: Disjoint Sets