COMP36111: Advanced Algorithms I

Lecture 2:

Almost Linear

Ian Pratt-Hartmann

Room KB2.38: email: ipratt@cs.man.ac.uk

2019–20
• In this lecture, we consider some basic algorithms operating on (undirected) graphs.

• The lecture is divided into four parts:
  • review of basic concepts and notation;
  • outline of the simple ‘union-find’ algorithm and its optimizations;
  • a mathematical interlude on very fast- (and slow-) growing functions;
  • analysis of the optimized union-find algorithm.

• There is also an appendix on random graphs: it’s not examinable, but I couldn’t resist.
Outline

Undirected graphs (revision)

Union-find structures

Fast- and slow-growing functions

Optimized union-find is almost linear

Appendix: Random graphs (not examinable)

Summary
• A graph is a pair $G = (V, E)$, where $V$ is a finite set and $E$ a set of subsets of $V$ of cardinality 2.
• We call the elements of $V$ vertices, and the elements of $E$ edges; we typically write edges $\{u, v\}$ as $(u, v)$.
• If $(u, v) \in E$, we say that $u$ and $v$ are neighbours.
• If $v \in e \in E$, we say $v$ and $e$ are adjacent.
• Graphs are typically displayed pictorially:

![Graph Diagram]

• We use the terms graph and undirected graph interchangeably.
The following are not pictures of graphs:

- Self-loops:
- Multiple edges
- Directions on edges
• Graphs, like directed graphs may be stored using either adjacency lists or adjacency matrices:
  • for adjacency lists, if vertex $i$ is in the adjacency list of vertex $j$, then vertex $j$ is in the adjacency list of vertex $i$;
  • for adjacency matrices, the matrix in question is always symmetric: $A_{i,j} = A_{j,i}$.

```
 v0

 v1

 v2

 v3
```

```
\begin{pmatrix}
  \ast & 1 & 0 & 1 \\
  1 & \ast & 1 & 1 \\
  0 & 1 & \ast & 1 \\
  1 & 1 & 1 & \ast \\
\end{pmatrix}
```

• In these lectures, we will employ adjacency lists by default.
• Any graph can be regarded as a directed graph in which all edges are bi-directional.
If $G = (V, E)$ is a graph, and $u, v \in V$, we say that $v$ is reachable from $u$ if there exists a sequence $u = u_0, \ldots, u_m = v$ from $V$ with $m \geq 0$ such that, for each $i$ ($0 \leq i < m$) $(u_i, u_{i+1}) \in E$. We call $u_0, \ldots, u_m$ a path of length $m$.

A graph is connected if every node is reachable from every other.

This notion gives rise to the following problem:

**CONNECTEDNESS**
Given: A graph $G = (V, E)$.
Return: Yes if $G$ is connected, No otherwise.

We can use DFS (regarding a graph as a directed graph) to test CONNECTEDNESS in time $O(|V| + |E|)$. 
• A **connected component** of a graph is a maximal set of vertices each of which is reachable from any other.

• It is easy to see that the connected components of a graph $G = (V, E)$ form a partition of $V$.

• Evidently, a graph is connected just in case it has exactly one connected component.

• This notion gives rise to the following task:

**CONNECTED COMPONENTS**
Given: A graph $G = (V, E)$.
Return: The connected components of $G$. 
• The following example illustrates the problem of finding the connected components of a graph.
The following example illustrates the problem of finding the connected components of a graph.
Outline

Undirected graphs (revision)

Union-find structures

Fast- and slow-growing functions

Optimized union-find is almost linear

Appendix: Random graphs (not examinable)

Summary
• Consider the task of finding the connected components of a graph.

• A partition of a (vertex) set $V$ is a collection of sets $P = \{A_1, \ldots, A_m\}$ such that:
  - $A_1 \cup \cdots \cup A_m = V$;
  - $A_i \cap A_j$ for all $i, j$, $(1 \leq i < j \leq m)$;
  - $A_i \neq \emptyset$ for all $i$ $(1 \leq i \leq m)$.

We refer to the elements of $P$ (sets of vertices) as cells.

• We create and manipulate $P$ using the following operations:
  - $\text{makeSet}(v)$—given a (pointer to) vertex $v$, add the singleton set $\{v\}$ to $P$.
  - $\text{union}(A_i, A_j)$—given (pointers to) the cells $A_i$ and $A_j$, remove these from $P$, and add the cell $A_i \cup A_j$.
  - $\text{find}(v)$—given a (pointer to) vertex $v$, return (a pointer to) the cell of $P$ containing $v$. 
• The algorithm for finding the connected components of $G = (V, E)$ is then simple:

```plaintext
find-cc(V,E)
    let P = ∅
    for ν ∈ V
        makeSet(ν)
    for (u, ν) ∈ E
        if find(u) ≠ find(ν)
            union(find(u), find(ν))
```

• This algorithm is obviously correct if the operations `makeSet`, `union` and `find` work as advertised: the connected components are the cells of the final partition.
• We consider first a naïve implementation of `makeSet`, `union` and `find`, where the cells of the partition are stored as lists.

• The code for `makeSet` is:

```
makeSet(ν)
    s = [ν] % (Create a new list)
    ν→ cell = s
    add s to P
```

observe that we add to each vertex a `cell`-pointer, containing the address of the cell it belongs to.

• The code for `find` is then simply:

```
find(ν)
    return ν→cell
```
• The code for union is just slightly more complicated:
  \[
  \text{union}(s, t) \\
  \text{remove } t \text{ from } P \\
  \text{for } v \in t \\
  \quad v \rightarrow \text{cell} = s \\
  s = \text{append}(s, t) \quad \text{\% note: this updates } P
  \]

• The append can be performed in constant time.

• However, we have to update the cell-pointers of all the vertices in \( t \), which takes time linear in \( t \).
• Thus, our algorithm

\[
\text{find-cc}(V,E) \\
\text{let } P = \emptyset \\
\text{for } v \in V \\
\quad \text{makeSet}(v) \\
\text{for } (u, v) \in E \\
\quad \text{if } \text{find}(u) \neq \text{find}(v) \\
\quad \quad \text{union}(\text{find}(u), \text{find}(v)) \\
\]

finds the connected components of \( G = (V, E) \), as required.

• This algorithm runs in time \( O(|V| \cdot |E|) \). Unless we are careful, it will also run in time \( \Omega(|V| \cdot |E|) \).
• There is a tiny, but very effective optimization we can perform:

\[
\text{when performing } \text{union}(s,t,P), \text{ make sure } |t| \leq |s|.
\]

• To do this, we assume that cells of the partition have their sizes attached:

\[
\text{makeSet}(v)
\]
\[
\begin{align*}
  s &= [v] \\
  s \rightarrow \text{size} &= 1 \\
  v \rightarrow \text{cell} &= s
\end{align*}
\]

• add \( s \) to \( P \)

\[
\text{union}(s,t)
\]
\[
\begin{align*}
  \text{remove } t \text{ from } P \\
  \text{for } v \in t \\
  v \rightarrow \text{cell} &= s \\
  s &= \text{append}(s,t) \quad \% \text{ note: this updates } P \\
  s \rightarrow \text{size} &= (s \rightarrow \text{size}) + (t \rightarrow \text{size})
\end{align*}
\]
Lemma
In a series of operations of `makeSet`, `union` and `find` on $n$ elements using the size-heuristic, no element can have its cell field assigned more than $\lceil \log n \rceil + 1$ times.

Proof.
Whenever $v \rightarrow \text{cell}$ changes, the cardinality of $v \rightarrow \text{cell}$ at least doubles. But $1 \leq |v \rightarrow \text{cell}| \leq n$. This can happen no more than $\lfloor \log n \rfloor$ times. $\Box$
Lemma

Using the size-heuristic, the algorithm `find-cc(G)` runs in time $O(n \log n)$, where $n = |G|$.

Proof.
The total number of calls to `makeSet`, `union` and `find` is linear in $|G|$. Each call involves a constant amount of work, apart from updating $v \rightarrow \text{cell}$. But each of the $\leq n$ vertices can be updated in this way at most $\lceil \log n \rceil$ times. Hence total work is $O(n) + O(n \log n) = O(n \log n)$. \qed
A further optimization is possible, by implementing the cells of the partition as trees.

Thus, we assume that each vertex has a field parent, which points to its parent in the tree, with the root pointing to itself.

We further assume that each vertex has a field rank, which records (roughly) the logarithm of the size of the tree rooted at that vertex.

Pointers to cells are just pointers to roots of trees.
We implement `makeSet` and `union` as

\[
\begin{align*}
\text{makeSet}(v) & \quad \text{union}(u, v, P) \\
& \quad v \rightarrow \text{parent} = v \\
& \quad \text{size}(v) = 1 \\
& \quad \text{size}(u) = \text{size}(u) + \text{size}(v) \\
& \quad \text{return } s
\end{align*}
\]

it being understood that \( u \) and \( v \) are roots and \( \text{size}(v) \leq \text{size}(u) \).
• We implement `makeSet` and `union` as

\[
\text{makeSet}(v) \quad \text{union}(u, v, P)
\]
\[
\begin{align*}
&v \rightarrow \text{parent} = v \\
&\text{size}(v) = 1 \\
&\text{return } s
\end{align*}
\]
\[
\begin{align*}
&v \rightarrow \text{parent} = u \\
&\text{size}(u) = \text{size}(u) + \text{size}(v)
\end{align*}
\]

it being understood that \( u \) and \( v \) are roots and \( \text{size}(v) \leq \text{size}(u) \).
• Implementing \texttt{find} just means following the parent-links to the root of the tree.

• The clever bit is that we flatten the tree in the process:
• Implementing `find` just means following the parent-links to the root of the tree.

• The clever bit is that we flatten the tree in the process:
• Implementing \texttt{find} just means following the parent-links to the root of the tree.
• The clever bit is that we flatten the tree in the process:
Implementing \texttt{find} just means following the parent-links to the root of the tree.

The clever bit is that we flatten the tree in the process:
• Implementing `find` just means following the parent-links to the root of the tree.

• The clever bit is that we flatten the tree in the process:
• Implementing `find` just means following the `parent`-links to the root of the tree.
• The clever bit is that we flatten the tree in the process:
• Implementing `find` just means following the `parent`-links to the root of the tree.

• The clever bit is that we flatten the tree in the process:
• Implementing \texttt{find} just means following the \texttt{parent}-links to the root of the tree.

• The clever bit is that we flatten the tree in the process:
The new code for `find` is:

```
find(v)
    if v →parent is not a root
        v →parent = find(v →parent)  %Flattening step
    return v →parent
```

That’s it. Now run the generic `find-cc` algorithm with these new implementations of `makeSet`, `union` and `find`.

The really fascinating aspect of this algorithm is the analysis of its running time. For that, we need some mathematical background . . .
Outline

Undirected graphs (revision)

Union-find structures

Fast- and slow-growing functions

Optimized union-find is almost linear

Appendix: Random graphs (not examinable)

Summary
• A polynomial (in $x$) is an expression $p(x)$ of the form

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0.$$  

• Such an expression defines a function $p : \mathbb{R} \to \mathbb{R}$ in the obvious way.

• A function $f : \mathbb{N} \to \mathbb{N}$ is polynomially bounded if, for some polynomial $p$,

  $$\forall n \ f(n) \leq p(n).$$

• The function $2^n$ is not polynomially bounded. Indeed, for any polynomial $p$,

  $$\exists n_0 \ \forall n \geq n_0 \ 2^n > p(n).$$
• A function \( f : \mathbb{N} \rightarrow \mathbb{N} \) is (singly) exponentially bounded if, for some polynomial \( p \),

\[
\forall n \ f(n) \leq 2^{p(n)}.
\]

• The function \( 2^{2^n} \) (\( = 2^{(2^n)} \)) is not exponentially bounded. Indeed, for any polynomial \( p \),

\[
\exists n_0 \ \forall n \geq n_0 \ 2^{2^n} > 2^{p(n)}.
\]

• A function \( f : \mathbb{N} \rightarrow \mathbb{N} \) is doubly exponentially bounded if, for some polynomial \( p \),

\[
\forall n \ f(n) \leq 2^{2^{p(n)}}.
\]

• A function \( f : \mathbb{N} \rightarrow \mathbb{N} \) is \( k \)-tuply exponentially bounded if, for some polynomial \( p \),

\[
\forall n \ f(n) \leq 2^{2^{\ldots^2} (p(n))} \Bigg\} k \text{ 2's}
\]
But that is not the end of the story. Consider the tower function

\[ t(n) = 2^{2^{\ldots^{2^n}}} \]

The function \( t(n) \) is not \( k \)-tuply exponentially bounded for any \( k \). Indeed, for all \( k \),

\[ \exists n_0 \ \forall n \geq n_0 \ \ t(n) > 2^{2^n} \ldots^{2^n} \]

The function \( t(n) \) is astonishingly fast-growing:

\[ t(5) = 2^{65,536} \]

(There are only about \( 2^{300} \) atoms in the universe.)
• To better understand the rates of growth of such functions, we use a simple language, LOOP, to compute them.

• LOOP has the following constructs:
  
  \[
  \begin{align*}
  &x = y, \quad x = 0, \quad x++, \quad \text{return } x, \\
  &\text{loop } (x) \{ \quad \cdots \quad \}
  \end{align*}
  \]

• Here is a LOOP program to compute \( x \mapsto 2x \):

  \[
  \begin{align*}
  &y_1 = x \\
  &\text{loop}(y_1)\{x++\} \\
  &\text{return } x
  \end{align*}
  \]
• Here is a LOOP program to compute $x \mapsto 2^x$:

```loop(y2)
\begin{align*}
y2 &= x \\
x &= 1 \\
\text{loop}(y2) \{ x &= 2x \} \\
\text{return } x
\end{align*}
```

• or, in other words, the program with doubly-embedded loops:

```loop(y2)
\begin{align*}
y2 &= x \\
x &= 1 \\
\text{loop}(y2) \{ \text{loop}(y1) \{ x &= x+1 \} \} \\
\text{return } x
\end{align*}
```
• Here is a LOOP program to compute \( x \mapsto t(x) \):

\[
y_3 = x \\
x = 1 \\
\text{loop}(y_3) \{ x = 2^x \} \\
\text{return } x
\]

• or, in other words, the program with triply-embedded loops:

\[
y_3 = x \\
x = 1 \\
\text{loop}(y_3) \{ 
  y_2 = x \\
  x = 1 \\
  \text{loop}(y_2) \{ 
    y_1 = x \\
    \text{loop}(y_1) \{ x++ \}
  \} \\
}\}
\text{return } x
\]
• We can go on. The program with quadruply-embedded loops:

\[
y_4 = x \\
x = 1 \\
\text{loop}(y_4) \{ x = t(x) \} \\
\text{return } x
\]

• computes a function \( f \) which grows with staggering rapidity:

\[
f(4) = 2^{16}. \]
• Define $\mathcal{L}_k$ to be the class of functions $\mathbb{N} \rightarrow \mathbb{N}$, that can be defined by programs in LOOP featuring at most $k$-tuple embeddings of the loop-constructor.

• Albert Meyers and Dennis Ritchie showed that

$$\mathcal{L}_1 \subsetneq \mathcal{L}_2 \subsetneq \mathcal{L}_3 \subsetneq \mathcal{L}_4 \cdots .$$

• Ignoring small values, this hierarchy coincides with a hierarchy defined earlier by Andrzej Grzegorczyk:

$$\mathcal{E}_1 \subsetneq \mathcal{E}_2 \subsetneq \mathcal{E}_3 \subsetneq \mathcal{E}_4 \cdots .$$

• The definition of the Grzegorczyk classes is complicated, but we don’t have to bother with it any more, because $\mathcal{E}_k = \mathcal{L}_{k-1}$ for $k \geq 3$. 
• Any function in $\mathcal{L}_2$ is said to be elementary.
• Any function in the collection
  \[
  \bigcup_{i=1}^{\infty} \mathcal{L}_i = \bigcup_{i=1}^{\infty} \mathcal{E}_i
  \]
  is said to be primitive recursive.
• Thus, the primitive recursive functions are those that can be computed by means of the LOOP language.
• Reality check: $t(n)$ is in $\mathcal{L}_3$. 
• But there are perfectly computable functions which grow more rapidly than any primitive recursive function.

• The best-known example is the function devised by Wilhelm Ackermann:

\[
A(m, n) = \begin{cases} 
n + 1 & \text{if } m = 0 \\
A(m - 1, 1) & \text{if } m > 0 \text{ and } n = 0 \\
A(m - 1, A(m, n - 1)) & \text{otherwise.}
\end{cases}
\]

You cannot compute, say, \( n \mapsto A(n, n) \) with \textsc{loop}: you need more general constructs (such as \textsc{while}).
• We shall use a slight variant of the Ackermann function. Define

\[ A_0(x) = x + 1 \]
\[ A_{i+1}(x) = A_i^{(x)}(x). \]

And, in addition, define

\[ A(x) = A_x(2). \]

• As an exercise, show that \( A_2(x) = 2^x \), and \( A_3(x) \geq t(x) \).
• In fact, \( A(x) \) is non-primitive-recursive.
• Just as we can define very fast-growing functions, we can define very slow-growing functions:

\[
\log^{(i)}(n) = \begin{cases} 
  n & \text{if } i = 0 \\
  \log(\log^{(i-1)}(n)) & \text{if } i > 0 \text{ and } \log^{(i-1)}(n) > 0 \\
  \text{undefined} & \text{otherwise}
\end{cases}
\]

• Now define \(\log^*(n) = \min\{i \geq 0 : \log^{(i)}(n) \leq 1\}\).

• It is easy to see that \(t\) and \(\log^*\) are (sort-of) inverses:

\[
\log^*(t(n)) = n.
\]

• Similarly, we can define an inverse of our variant of Ackermann’s function: \(\alpha(n) = \min\{i \geq 0 : A(i) \geq n\}\).

• To all intents and purposes, \(\log^*(n) \leq 5\) and \(\alpha(n) \leq 4\).
Outline

Undirected graphs (revision)

Union-find structures

Fast- and slow-growing functions

Optimized union-find is almost linear

Appendix: Random graphs (not examinable)

Summary
Theorem

Using the tree implementation, the algorithm `find-cc(G)` runs in time $O(n\alpha(n))$, where $n = |G|$.

Proof.
Define $n(v)$ to be the number of nodes in the tree rooted at $v$ assuming no path compression.

$$r(v) = \lceil \log n(v) \rceil + 2.$$ 

Call $r(v)$ the rank of vertex $v$. Denote parent of $v$ by $p(v)$.

1. $r(v) \leq \lceil \log n \rceil + 2$.
2. If $p(v) = w$, then $r(v) < r(w)$.
3. The number of vertices with rank $s$ is at most $n/(2^{s-2})$.

Continued on next slide . . .
Proof (contd.)

Suppose we carry out \( m \) makeSet, union and find operations on a collection of \( n \) vertices. Each call to makeSet or union requires a constant number of operations. We need only worry about the work done by the calls to find.

The tree-structure changes as we modify the pointers (path compression). Let the parent of \( v \) after \( t \) pointer modifications be denoted \( p^t(v) \).

We give each vertex \( v \) a label during the evolving computation:

\[
L^t(v) = \text{the largest } i \text{ s.t. } r(p^t(v)) \geq A_i(r(v)).
\]

Continued on next slide . . .
Proof (contd.)

For $n > 5$, for a non-root vertex $v$ and setting $i = L^t(v)$,

$$n > \lceil \log n \rceil + 2 \geq r(p^t(v)) \geq A_i(r(v)) \geq A_i(2)$$

whence $L^t(v) < \alpha(n)$ for all $v$ and $t$.

Consider a vertex $v$ in the union-find graph. We charge £1 whenever the parent-pointer of $v$ is moved at time $t$. We count the total charge. Let the root of the tree containing $v$ at time $t$ be $z$.

- if $v$ has a proper ancestor not equal to $z$ such that $L^t(v) = L^t(w)$, charge £1 to $v$;
- otherwise, charge £1 to the current invocation of find.

How much charge can accrue?

Continued on next slide . . .
Proof (contd.)

The maximum charge accruing to any invocation of \texttt{find}(u) is the number of distinct values \(L^t(v)\) on the path from \(u\) to its root.

This is at most \(\alpha(n)\). Hence the total changes assigned to all \(m\) \texttt{find}-operations is at most \(m\alpha(n)\).

Continued on next slide...
Proof (contd.)

What is the total charge accruing to any vertex \( v \)?

Suppose \( £1 \) is charged to \( v \) at time \( t \). Then \( v \) has a proper ancestor \( w \) (not the root) with \( L^t(v) = L^t(w) = i \).

Notice that \( r(p^t(v)) \geq A_i(r(v)) \) and \( r(p^t(w)) \geq A_i(r(w)) \).

Thus there exists \( k \geq 1 \) such that \( r(p^t(v)) \geq A_i^{(k)}(r(v)) \).

Then

\[
r(z) \geq r(p^t(w)) \geq A_i(r(w)) \geq A_i(r(p^t(v))) \geq A_i(A_i^{(k)}(r(v))) = A_i^{(k+1)}(r(v)).
\]

Continued on next slide . . .
Proof (contd.)

So we have shown that \( r(z) \geq A_i^{(k+1)}(r(v)) \).

But now the parent of \( v \) becomes \( z \), so \( r(p^{t+1}(v)) \geq A_i^{(k+1)}(r(v)) \).

Hence, \( v \) can be charged \( \£ 1 \) at most \( r(v) \) times before we reach a time \( t' \) when \( r(p^{t'}(v)) \geq A_i^{r(v)}(r(v)) = A_{i+1}(r(v)) \), whence \( L^{t'}(v) \geq i + 1 > L^t(v) \).

Continued on next slide . . .
Proof (contd.)

But $L^t(v)$ can take at most $\alpha(n)$ values (at $t$ changes), whence $v$ can certainly accrue a charge of at most $\xi r(v) \cdot \alpha(n)$. That is, at most $\xi s \cdot \alpha(n)$ can be charged to any vertex of rank $s$ ($0 \leq s < \log n + 2$).

There are at most $n/2^{s-2}$ vertices with rank $s$.

So total charges accruing to vertices over the course of all find-calls is

$$
\sum_{s=0}^{\lfloor \log n \rfloor + 2} s\alpha(n) \frac{n}{2^{s-2}} \leq n\alpha(n) \left( \sum_{s=0}^{\infty} \frac{s}{2^{s-2}} \right) \leq 8n\alpha(n).
$$
Outline

Undirected graphs (revision)

Union-find structures

Fast- and slow-growing functions

Optimized union-find is almost linear

Appendix: Random graphs (not examinable)

Summary
• Suppose we take a collection $V$ of $n$ vertices, and set $E = \emptyset$.

• Fix $p$ ($0 \leq p \leq 1$).

• For each pair of (distinct) vertices, $u$, $v$ add the edge $(u, v)$ to $E$ with probability $p$.

• We then obtain a graph. E.g., for $n = 10$ and $p = 0.2$, we might obtain the following graph (given in adjacency-list form):

0: 5:
1: 3
2:
3: 1 4
4: 3 6 8 9

5: 6: 4 8
6: 7: 8
7: 8: 4 6 7
8: 9: 4

• This graph is not connected, but we might have generated a connected graph!
• We can think of the graph obtained in this way, with $n$ vertices and (independent) connection probability for each edge $p$, as a random variable, $G_{p,n}$.

• We call $G_{p,n}$ a *random graph*.

• By sampling a number of such graphs and running the test for connectedness, we can estimate this probability that $G_{p,n}$ is connected.
The results are rather surprising:

We see from these graphs:

- The probability of connectedness is very close to 0 or 1 outside a tiny window (look at the X-axis).
- That window gets narrower as $n$ increases.
• Also slightly surprising is what happens when the graph is not connected.

• What distributions of sizes to you expect for the connected components?

• Consider our earlier randomly generated graph:

0: 5:
1: 3  6: 4
2:    7: 8
3: 1 4 8: 4 6 7
4: 3 6 8 9 9: 4

• This graph has 4 connected components, namely 
  \{\{0\}, \{1, 3, 4, 6, 7, 8, 9\}, \{2\}, \{5\}\}.

• Notice how one component (the so-called giant component) is much bigger than the others. Can you see why, on reflection, this is to be expected?
Outline

Undirected graphs (revision)

Union-find structures

Fast- and slow-growing functions

Optimized union-find is almost linear

Appendix: Random graphs (not examinable)

Summary
• Summary
  • The concepts connected and connected component for graphs;
  • the union-find algorithm;
  • the Grzegorczyk hierarchy of primitive recursive functions, the language LOOP and the Ackermann function, $A(m, n)$;
  • the $O((m + n)\alpha(n))$ upper complexity bound on (optimized) union-find.

• Reading:
  • G+T Ch 13 to end of 13.4.1, pp. 353–377.
  • G+T: Ch 7, pp. 219–235.