COMP36111: Advanced Algorithms I

Lecture 1b:

Connected components of Undirected Graphs, Fast- and Slow-growing Functions

Ian Pratt-Hartmann

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

2017–18
Outline

Union-find structures

Fast- and slow-growing functions

An almost-linear solution

Random graphs (not examinable)
- Recall the problem of finding connected components of an undirected graph.
Recall the problem of finding connected components of an undirected graph.
• We construct a partition $P = \{A_1, \ldots, A_m\}$ on the vertex set $V$:
  • $A_1 \cup \cdots \cup A_m = V$;
  • $A_i \cap A_j$ for all $i, j$, $(1 \leq i < j \leq m)$;
  • $A_i \neq \emptyset$ forall $i$ $(1 \leq i \leq m)$.
• 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 $v \in V$
        makeSet($v$)
    for $(u, v) \in E$
        if find($u$) ≠ find($v$)
            union(find($u$), find($v$))
```

• 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:

```plaintext
makeSet(v)
    s= [v] % (Create a new list)
    v→ cell= s
    return s
```

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:

```plaintext
find(v)
    return v →cell
```
• The code for union is just slightly more complicated:

```python
union(s,t,P)
    P= remove(t,P)
    for v ∈ t
        v → cell= s
    s= append(s,t)
    return s
```

• 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.
• There is a tiny, but very effective optimization we can perform:

```plaintext
when performing union(s,t,P), make sure |t| ≤ |s|.
```

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

```plaintext
makeSet(v)
    s = [v]
    s → size = 1
    v → cell = s
    return s

union(s,t,P)
    P = remove(t,P)
    for v ∈ t
        v → cell = s
    s = append(s,t)
    s → size = (s → size) + (t → size)
    return s
```
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 $\lfloor \log n \rfloor + 1$ 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.
Lemma

*Using the size-heuristic, the algorithm* \text{find-cc}(G) *runs in time* \( O(n \log n) \), *where* \( n = \lvert G \rvert \).

Proof.

The total number of calls to \text{makeSet}, \text{union} and \text{find} is linear in \( \lvert G \rvert \). 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 \( \lfloor \log n \rfloor \) times. Hence total work is \( O(n) + O(n \log n) = O(n \log n) \). \( \square \)
Outline

Union-find structures

Fast- and slow-growing functions

An almost-linear solution

Random graphs (not examinable)
• 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 \(p\),

\[
\forall n \quad f(n) \leq p(n).
\]

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

\[
\exists n_0 \quad \forall n \geq n_0 \quad 2^n > p(n).
\]
- A function $f : \mathbb{N} \to \mathbb{N}$ is (singly) exponentially bounded if, for some polynomial $p$,
\[ \forall n \quad f(n) \leq 2^{p(n)}. \]

- The function $2^{2^n}$ ($= 2^{(2^n)}$) is not polynomially bounded. Indeed, for any polynomial $p$,
\[ \exists n_0 \quad \forall n \geq n_0 \quad 2^{2^n} > 2^{p(n)}. \]

- A function $f : \mathbb{N} \to \mathbb{N}$ is doubly exponentially bounded if, for some polynomial $p$,
\[ \forall n \quad f(n) \leq 2^{2^{p(n)}}. \]

- A function $f : \mathbb{N} \to \mathbb{N}$ is $k$-tuply exponentially bounded if, for some polynomial $p$,
\[ \forall n \quad f(n) \leq 2^{2^{2^{\cdots^{2^{p(n)}}}}}} \text{ ($k$ \ 2's)} \]
• But that is not the end of the story. Consider the tower function

\[ t(n) = 2 \underbrace{2^2 \cdots 2^2}_n \text{ }'s \].

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

\[ \exists n_0 \quad \forall n \geq n_0 \quad t(n) > 2 \underbrace{2^2 \cdots 2^2}_{k \text{ }'s} \].

• 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:
  
  \[
  x = y, \quad x = 0, \quad x++, \quad \text{return} \ x, \\
  \text{loop} (x) \{ \quad \cdots \quad \}
  \]

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

  ```
  \text{y1} = x \\
  \text{loop(y1)} \{ x++ \} \\
  \text{return} \ x
  ```
• Here is a LOOP program to compute $x \mapsto 2^x$:

```loop(y2)
  x = 2x
return x
```

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

```loop(y2)
  y1 = x
  loop(y1)
    x++
return x
```
• 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:

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

\begin{verbatim}
y4= x
x= 1
loop(y4){x= t(x)}
return x
\end{verbatim}

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

\[
 f(4) = 2^{2^{\cdots^{2^{16}}}} .
\]
• 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:

\[ \mathcal{E}_1 \subsetneq \mathcal{E}_2 \subsetneq \mathcal{E}_3 \subsetneq \mathcal{L}_4 \cdots. \]

• 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 $L_2$ is said to be elementary.
• Any function in the collection

$$\bigcup_{i=1}^{\infty} L_i = \bigcup_{i=1}^{\infty} E_i$$

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 $L_3$. 
• But there are perfectly computable functions which grow more rapidly than any primitive recursive function.

• The best-known example is the Ackermann function:

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

You cannot compute, say, \( n \mapsto A(n, n) \) with

• LOOP: you need more general constructs (such as 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:

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

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

• 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, \( \alpha(n) \leq 4 \).
Outine

Union-find structures

Fast- and slow-growing functions

An almost-linear solution

Random graphs (not examinable)
• A further optimization is possible, by implementing the cells of the partition as trees.
• Thus, we assume that each vertex has a field mother, which points to its mother 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 \texttt{makeSet} and \texttt{union} as

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

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

\[
\text{makeSet}(v) \quad \text{union}(u,v,P)
\]

\[
v \rightarrow \text{mother} = v \quad v \rightarrow \text{mother} = u
\]

\[
\text{size}(v) = 1 \quad \text{size}(u) = \text{size}(u) + \text{size}(v)
\]

return s \quad \text{return s}

it being understood that \( \text{size}(v) \leq \text{size}(u) \).
• Implementing `find` just means following the mother-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 mother-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 mother-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 mother-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 `mother`-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 **mother**-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 `mother-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 mother-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:

```python
find(v)
    if v → mother is not a root
        v → mother = find(v → mother)  # Flattening step
    return v → mother
```

- 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.
- We have the following result, due to R.E. Tarjan:
Theorem

*Using the tree implementation, the algorithm* \( \text{find-cc}(G) \) *runs in time* \( O(n^{\alpha(n)}) \), *where* \( n = |G| \).

We omit the proof of this theorem from the course. But it is comprehensible. See Goodrich and Tamassia pp. 233 ff.
Outline

Union-find structures

Fast- and slow-growing functions

An almost-linear solution

Random graphs (not examinable)
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
  7: 8
  8: 4 6 7
  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:

![Graph](image)

- X-axis: $p$
- Y-axis: prob. $G_{p,n}$ connected

$n = 800 \ (800)$
$n = 400$
$n = 300$
$n = 200$
$n = 100$

• 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
2: 7: 8
3: 1 4
4: 3 6 8 9
5: 6: 4 8
    7: 8
    8: 4 6 7
    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?
Summary

• In this lecture, we have considered:
  • an $O(n)$ algorithm for determining graph connectedness;
  • the Grzegorczyk hierarchy of primitive recursive functions, the language LOOP and the Ackermann function $A(m, n)$;
  • an $O(n^\alpha(n))$ algorithm for computing connected components;
  • random graphs.

Items marked in blue are not examinable.

• Reading:
  • G+T Ch 13 to end of 13.4.1, pp. 353–377.
  • G+T: Ch 7, pp. 219–239.