Introduction to Java (cs2514)

Lecture 17: Bloom Filters

M. R. C. van Dongen

March 17, 2017

Introduction

- This lecture studies *bitmaps* and *Bloom Filters*.
- Bitmaps are space-optimal data structures for representing sets.
- Candidate set members come from an index set $\{0, \ldots, n-1\}$.
- They can answer simple questions about the set:
  - Iis a candidate member in the set?
- *Bloom Filters* are also used to represent sets.
- Here the members come from a key set $\{k_0, \ldots, k_{n-1}\}$.
- The answers from Bloom Filters are *probabilistic*:
  - Not all answers are correct with 100% probability.
- When n gets large, we cannot represent such sets as bitmaps.
- However, Bloom Filters for such sets are small.
- They require only a few bits per element.

Bitmaps

- A *bitmap* represents the members of a given index set.
- Here an index set is a set of the form $\{0, \ldots, n-1\}$.
- A bitmap may be represented using a `boolean` array.
- `Java` does not prescribe how represent a `boolean`.
- If a `boolean` is represented as an `int` then this wastes memory.
- Especially if the index size is large.
- In practice bitmaps are often represented as `int` arrays.

Basic Representation

`Bitmap( int capacity )`: Create a bitmap.
`add( int index )`: Add a given index.
`remove( int index )`: Remove a given index.
`size( )`: What is the cardinality of the bitmap?
`contains( int index )`: Does the bitmap contains an index?

Implementation

Java

```
public class Bitmap {
    private final boolean[] bits;
    private int size;

    public Bitmap( int capacity ) {
        bits = new boolean[ capacity ];
        size = 0;
    }

    public void add( int index ) {
        size += (bits[ index ]) ? 0 : 1;
        bits[ index ] = true;
    }

    public void remove( int index ) {
        size -= (bits[ index ]) ? 1 : 0;
        bits[ index ] = false;
    }

    public int size( ) {
        return size;
    }

    public boolean contains( int index ) {
        return bits[ index ];
    }
}
```

Implementation

Java

```
public class Bitmap {
    private final boolean[] bits;
    private int size;

    public Bitmap( int capacity ) {
        bits = new boolean[ capacity ];
        size = 0;
    }

    public void add( int index ) {
        size += (bits[ index ]) ? 0 : 1;
        bits[ index ] = true;
    }

    public void remove( int index ) {
        size -= (bits[ index ]) ? 1 : 0;
        bits[ index ] = false;
    }

    public int size( ) {
        return size;
    }

    public boolean contains( int index ) {
        return bits[ index ];
    }
}
```

Implementation

Java

```
public class Bitmap {
    private final boolean[] bits;
    private int size;

    public Bitmap( int capacity ) {
        bits = new boolean[ capacity ];
        size = 0;
    }

    public void add( int index ) {
        size += (bits[ index ]) ? 0 : 1;
        bits[ index ] = true;
    }

    public void remove( int index ) {
        size -= (bits[ index ]) ? 1 : 0;
        bits[ index ] = false;
    }

    public int size( ) {
        return size;
    }

    public boolean contains( int index ) {
        return bits[ index ];
    }
}
```

Implementation

Java

```
public class Bitmap {
    private final boolean[] bits;
    private int size;

    public Bitmap( int capacity ) {
        bits = new boolean[ capacity ];
        size = 0;
    }

    public void add( int index ) {
        size += (bits[ index ]) ? 0 : 1;
        bits[ index ] = true;
    }

    public void remove( int index ) {
        size -= (bits[ index ]) ? 1 : 0;
        bits[ index ] = false;
    }

    public int size( ) {
        return size;
    }

    public boolean contains( int index ) {
        return bits[ index ];
    }
}
```

Implementation

Java

```
public class Bitmap {
    private final boolean[] bits;
    private int size;

    public Bitmap( int capacity ) {
        bits = new boolean[ capacity ];
        size = 0;
    }

    public void add( int index ) {
        size += (bits[ index ]) ? 0 : 1;
        bits[ index ] = true;
    }

    public void remove( int index ) {
        size -= (bits[ index ]) ? 1 : 0;
        bits[ index ] = false;
    }

    public int size( ) {
        return size;
    }

    public boolean contains( int index ) {
        return bits[ index ];
    }
}
```

Simulation

Java

```java
Bitmap set = new Bitmap( 10 );
set.add( 4 );
set.add( 8 );
set.add( 1 );
set.remove( 8 );
set.add( 4 );
```

Simulation

Java

```
Bitmap set = new Bitmap( 10 );
set.add( 4 );
set.add( 8 );
set.add( 1 );
set.remove( 8 );
set.add( 4 );
```

size:0

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

Simulation

Java

```
Bitmap set = new Bitmap( 10 );
set.add( 4 );
set.add( 8 );
set.add( 1 );
set.remove( 8 );
set.add( 4 );
```

size: 1

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

Simulation

Java

```java
Bitmap set = new Bitmap( 10 );
set.add( 4 );
set.add( 8 );
set.add( 1 );
set.remove( 8 );
set.add( 4 );
```

size: 2

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

Simulation

Java

```
Bitmap set = new Bitmap( 10 );
set.add( 4 );
set.add( 8 );
set.add( 1 );
set.remove( 8 );
set.add( 4 );
```

size: 3

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

Simulation

Java

```
Bitmap set = new Bitmap( 10 );
set.add( 4 );
set.add( 8 );
set.add( 1 );
set.remove( 8 );
set.add( 4 );
```

size: 2

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

Simulation

Java

```
Bitmap set = new Bitmap( 10 );
set.add( 4 );
set.add( 8 );
set.add( 1 );
set.remove( 8 );
set.add( 4 );
```

size: 2

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

Using int Representation

- Remember a Java int is implemented with 32 bits.
- We may represent a bitmap with capacity of n using $\lceil n/32 \rceil$ ints.
- The Integer class defines the number of bits in an int:
  - Integer.SIZE.

Java

```
private final int[] bits;
private int size;
public Bitmap( int capacity ) {
    bits = new int[ (capacity + Integer.SIZE - 1) / Integer.SIZE ];
    size = 0;
}
```

- We use bit n to represent index n.

Using int Representation

- Remember a Java int is implemented with 32 bits.
- We may represent a bitmap with capacity of n using $\lceil n/32 \rceil$ ints.
- The Integer class defines the number of bits in an int:
  - Integer.SIZE.

Java

```
private final int[] bits;
private int size;
public Bitmap( int capacity ) {
    bits = new int[ (capacity + Integer.SIZE - 1) / Integer.SIZE ];
    size = 0;
}
```

- We use bit n to represent index n.
- But how do we turn the bits on and off?

Manipulating Bits

- Implementing bit operations may be done with arithmetic.
- For example, let `bits` be an `int`.
- Then `(bits / 1) % 2` is the zeroth (least significant) bit.
- Then `(bits / 2) % 2` is the first bit.
- Then `(bits / 4) % 2` is the second bit.
- Then `(bits /` 2^n`) % 2` is the nth bit.

Manipulating Bits

- Implementing bit operations may be done with arithmetic.
- For example, let `bits` be an `int`.
- Then `(bits / 1) % 2` is the zeroth (least significant) bit.
- Then `(bits / 2) % 2` is the first bit.
- Then `(bits / 4) % 2` is the second bit.
- Then `(bits /` 2^n`) % 2` is the nth bit.
- `bits - (((bits /` 2^n`) % 2) *` 2^n`)` turns the nth bit off.

Manipulating Bits

- Implementing bit operations may be done with arithmetic.
- For example, let `bits` be an `int`.
- Then `(bits / 1) % 2` is the zeroth (least significant) bit.
- Then `(bits / 2) % 2` is the first bit.
- Then `(bits / 4) % 2` is the second bit.
- Then `(bits /` 2^n`) % 2` is the nth bit.
- `bits - (((bits /` 2^n`) % 2) *` 2^n`)` turns the nth bit off.
- `bits + ((1 - (bits /` 2^n`) % 2) *` 2^n`)` turns the bit on.

Manipulating Bits (Continued)

Java

```java
final static int SHIFT = Integer.numberOfTrailingZeros( Integer.SIZE );
final static int MASK  = Integer.SIZE - 1;
public boolean contains( int index ) {
    return (bits[ index >> SHIFT ] & (1 << (index & MASK))) != 0;
}
```

Bitwise Operations: Not Examinable

Operator	Example	Result	Description
`a >> b`	`32 >> 4`	2	Shift `a` right by `b` bits. (Keep sign.)
`a >>> b`	`-1 >>> 30`	3	Shift `a` right by `b` bits. (Reset sign.)
`a << b`	`3 << 3`	24	Shift `a` left by `b` bits.
`a & b`	`5 & 3`	1	Bitwise and of `a` and `b`.
`a | b`	`5 | 3`	7	Bitwise or of `a` and `b`.
`a ^ b`	`5 ^ 3`	6	Bitwise exclusive or of `a` and `b`.
`~a`	`~0`	-1	Bitwise complement of `a`.

Some Facts

- An n-bit bitmap may be represented with $(n+7)/8$ bytes.
- We can represent all possible 2^n subset configurations.
- All operations take constant time.
- All operations are correct: there are no errors.

When Big Gets Huge

- Bitmaps are nice if your sets are of the form $\{0, \ldots, n-1\}$.
- Many applications rely on sets U, with $n = |U|$, but
  - $n \ll \max_{u \in U}(u)$.
- We could still represent them as bitmaps.
- However, this would require $\max_{u \in U}(u)$ bits.
- Should this happen, a lot of time is wasted on swopping.
- A datastructure that fits in to memory would be *much* faster:
  - Even if the resulting "set" operations are sometimes inaccurate.

Introduction to Bloom Filters

- A *Bloom Filter* is a probabilistic version of a “set.”
- It requires much less memory than a bitmap-based set.
- It cannot answer questions about size.
- You can add to the set, but you cannot remove from the set.
- It can be used to answer set membership queries:
  - `contains( int key )`?
  - The accuracy of the answer depends on the answer:

    `false`: Set definitely doesn’t contain `key`.

    `true`: Set contains `key` with a certain degree of confidence.

Introduction to Bloom Filters (Continued)

- Let u be a random member from the set member universe U.
- Furthermore, let B be a Bloom Filter.
- There are four scenarios.
  - B returns `false` and $u \notin B$: u is definitely not in B.
  - B returns `false` and $u \in B$: this situation cannot occur.
  - B returns `true` and $u \in B$: $u \in B$.
  - B returns `true` and $u \notin B$: *False positive*. (May happen).

Introduction to Bloom Filters (Continued)

- Let u be a random member from the set member universe U.
- Furthermore, let B be a Bloom Filter.
- There are four scenarios.
  - B returns `false` and $u \notin B$: u is definitely not in B.
  - B returns `false` and $u \in B$: this situation cannot occur.
  - B returns `true` and $u \in B$: $u \in B$.
  - B returns `true` and $u \notin B$: *False positive*. (May happen).

Introduction to Bloom Filters (Continued)

- Let u be a random member from the set member universe U.
- Furthermore, let B be a Bloom Filter.
- There are four scenarios.
  - B returns `false` and $u \notin B$: u is definitely not in B.
  - B returns `false` and $u \in B$: this situation cannot occur.
  - B returns `true` and $u \in B$: $u \in B$.
  - B returns `true` and $u \notin B$: *False positive*. (May happen).

Introduction to Bloom Filters (Continued)

- Let u be a random member from the set member universe U.
- Furthermore, let B be a Bloom Filter.
- There are four scenarios.
  - B returns `false` and $u \notin B$: u is definitely not in B.
  - B returns `false` and $u \in B$: this situation cannot occur.
  - B returns `true` and $u \in B$: $u \in B$.
  - B returns `true` and $u \notin B$: *False positive*. (May happen).

Introduction to Bloom Filters (Continued)

- Let u be a random member from the set member universe U.
- Furthermore, let B be a Bloom Filter.
- There are four scenarios.
  - B returns `false` and $u \notin B$: u is definitely not in B.
  - B returns `false` and $u \in B$: this situation cannot occur.
  - B returns `true` and $u \in B$: $u \in B$.
  - B returns `true` and $u \notin B$: *False positive*. (May happen).
- In general we don't know whether $u \in B$ or $u \notin B$.
  - (That's why we're using B.)

Introduction to Bloom Filters (Continued)

- Let u be a random member from the set member universe U.
- Furthermore, let B be a Bloom Filter.
- There are four scenarios.
  - B returns `false` and $u \notin B$: u is definitely not in B.
  - B returns `false` and $u \in B$: this situation cannot occur.
  - B returns `true` and $u \in B$: $u \in B$.
  - B returns `true` and $u \notin B$: *False positive*. (May happen).
- In general we don't know whether $u \in B$ or $u \notin B$.
  - (That's why we're using B.)
- We never have to worry when B returns `false`.
- We only have to be careful that false positives may occur.
- So when B returns `true`, the result may be incorrect.

Observations

- Let's revisit our bitmaps.
- We may view a bitmap as a key-value system.
- The keys and values are equal.
- With perfect hashing we could decide set membership of n members using n bits:
  - We hash each set members to its hash code.
  - The hash codes should be in the range $\{0, \ldots, n-1\}$.
  - Use the hash code as an index in the array.
- Unfortunately, collisions are very common.

Implementation

- Let U be the universe of candidate set members.
- Let $n = |U|$ and let $I = \{0, \ldots, m-1\}$, with $m \geq n$.
- With one *perfect* hash function, $h_0 : U \rightarrow I$ we can decide set membership with a bitmap.
- Bloom Filters decide set membership with *several* hash functions, $h_i : U \rightarrow I$.
- An empty filter is represented with m bits: each bit is 0.
- To add $u \in U$, we set Bit $h_i(u)$ to 1 for each hash function $h_i(\cdot)$.
- Then u is not contained if Bit $h_i(u)$ is 0 for some hash function.
- Otherwise, u is in the set, but false positives may occur.

Implementation

- For example, let's assume $U = \{0, 1, 7\}$.
- Furthermore, let's use 4 bits and 2 hash functions:
  - $h_0(k) = k + 1 \bmod 3$ and $h_1(k) = k \bmod 4$.

Implementation

- For example, let's assume $U = \{0, 1, 7\}$.
- Furthermore, let's use 4 bits and 2 hash functions:
  - $h_0(k) = k + 1 \bmod 3$ and $h_1(k) = k \bmod 4$.

k	$h_0(k)$	$h_1(k)$	**Union of Bits**

Implementation

- For example, let's assume $U = \{0, 1, 7\}$.
- Furthermore, let's use 4 bits and 2 hash functions:
  - $h_0(k) = k + 1 \bmod 3$ and $h_1(k) = k \bmod 4$.

k	$h_0(k)$	$h_1(k)$	**Union of Bits**
0	1	0	0011

Implementation

- For example, let's assume $U = \{0, 1, 7\}$.
- Furthermore, let's use 4 bits and 2 hash functions:
  - $h_0(k) = k + 1 \bmod 3$ and $h_1(k) = k \bmod 4$.

k	$h_0(k)$	$h_1(k)$	**Union of Bits**
0	1	0	0011
1	2	1	0110

Implementation

- For example, let's assume $U = \{0, 1, 7\}$.
- Furthermore, let's use 4 bits and 2 hash functions:
  - $h_0(k) = k + 1 \bmod 3$ and $h_1(k) = k \bmod 4$.

k	$h_0(k)$	$h_1(k)$	**Union of Bits**
0	1	0	0011
1	2	1	0110
7	2	3	1100

Implementation

- For example, let's assume $U = \{0, 1, 7\}$.
- Furthermore, let's use 4 bits and 2 hash functions:
  - $h_0(k) = k + 1 \bmod 3$ and $h_1(k) = k \bmod 4$.

k	$h_0(k)$	$h_1(k)$	**Union of Bits**
0	1	0	0011
1	2	1	0110
7	2	3	1100

Keys Added	**Bits Set**	$0 \in B$?	$1 \in B$?	$7 \in B$?

Implementation

- For example, let's assume $U = \{0, 1, 7\}$.
- Furthermore, let's use 4 bits and 2 hash functions:
  - $h_0(k) = k + 1 \bmod 3$ and $h_1(k) = k \bmod 4$.

k	$h_0(k)$	$h_1(k)$	**Union of Bits**
0	1	0	0011
1	2	1	0110
7	2	3	1100

Keys Added	**Bits Set**	$0 \in B$?	$1 \in B$?	$7 \in B$?
$\{\,\}$	0000	–	–	–

Implementation

- For example, let's assume $U = \{0, 1, 7\}$.
- Furthermore, let's use 4 bits and 2 hash functions:
  - $h_0(k) = k + 1 \bmod 3$ and $h_1(k) = k \bmod 4$.

k	$h_0(k)$	$h_1(k)$	**Union of Bits**
0	1	0	0011
1	2	1	0110
7	2	3	1100

Keys Added	**Bits Set**	$0 \in B$?	$1 \in B$?	$7 \in B$?
$\{\}$	0000	−	−	−
$\{0\}$	0011	+	−	−

Implementation

- For example, let's assume $U = \{0, 1, 7\}$.
- Furthermore, let's use 4 bits and 2 hash functions:
  - $h_0(k) = k + 1 \bmod 3$ and $h_1(k) = k \bmod 4$.

k	$h_0(k)$	$h_1(k)$	**Union of Bits**
0	1	0	0011
1	2	1	0110
7	2	3	1100

Keys Added	**Bits Set**	$0 \in B$?	$1 \in B$?	$7 \in B$?
$\{\,\}$	0000	−	−	−
$\{0\}$	0011	+	−	−
$\{1\}$	0110	−	+	−

Implementation

- For example, let's assume $U = \{0, 1, 7\}$.
- Furthermore, let's use 4 bits and 2 hash functions:
  - $h_0(k) = k + 1 \bmod 3$ and $h_1(k) = k \bmod 4$.

k	$h_0(k)$	$h_1(k)$	**Union of Bits**
0	1	0	0011
1	2	1	0110
7	2	3	1100

Keys Added	**Bits Set**	$0 \in B$?	$1 \in B$?	$7 \in B$?
$\{\}$	0000	−	−	−
$\{0\}$	0011	+	−	−
$\{1\}$	0110	−	+	−
$\{7\}$	1100	−	−	+

Implementation

- For example, let's assume $U = \{0, 1, 7\}$.
- Furthermore, let's use 4 bits and 2 hash functions:
  - $h_0(k) = k + 1 \bmod 3$ and $h_1(k) = k \bmod 4$.

k	$h_0(k)$	$h_1(k)$	**Union of Bits**
0	1	0	0011
1	2	1	0110
7	2	3	1100

Keys Added	**Bits Set**	$0 \in B$?	$1 \in B$?	$7 \in B$?
$\{\}$	0000	−	−	−
$\{0\}$	0011	+	−	−
$\{1\}$	0110	−	+	−
$\{7\}$	1100	−	−	+
$\{0, 1\}$	0111	+	+	−

Implementation

- For example, let's assume $U = \{0, 1, 7\}$.
- Furthermore, let's use 4 bits and 2 hash functions:
  - $h_0(k) = k + 1 \bmod 3$ and $h_1(k) = k \bmod 4$.

k	$h_0(k)$	$h_1(k)$	**Union of Bits**
0	1	0	0011
1	2	1	0110
7	2	3	1100

Keys Added	**Bits Set**	$0 \in B$?	$1 \in B$?	$7 \in B$?
$\{\}$	0000	−	−	−
$\{0\}$	0011	+	−	−
$\{1\}$	0110	−	+	−
$\{7\}$	1100	−	−	+
$\{0, 1\}$	0111	+	+	−
$\{0, 7\}$	1111	+	+	+

Implementation

- For example, let's assume $U = \{0, 1, 7\}$.
- Furthermore, let's use 4 bits and 2 hash functions:
  - $h_0(k) = k + 1 \bmod 3$ and $h_1(k) = k \bmod 4$.

k	$h_0(k)$	$h_1(k)$	**Union of Bits**
0	1	0	0011
1	2	1	0110
7	2	3	1100

Keys Added	**Bits Set**	$0 \in B$?	$1 \in B$?	$7 \in B$?
$\{\}$	0000	−	−	−
$\{0\}$	0011	+	−	−
$\{1\}$	0110	−	+	−
$\{7\}$	1100	−	−	+
$\{0, 1\}$	0111	+	+	−
$\{0, 7\}$	1111	+	+	+
$\{1, 7\}$	1110	−	+	+

Implementation

- For example, let's assume $U = \{0, 1, 7\}$.
- Furthermore, let's use 4 bits and 2 hash functions:
  - $h_0(k) = k + 1 \bmod 3$ and $h_1(k) = k \bmod 4$.

k	$h_0(k)$	$h_1(k)$	**Union of Bits**
0	1	0	0011
1	2	1	0110
7	2	3	1100

Keys Added	**Bits Set**	$0 \in B$?	$1 \in B$?	$7 \in B$?
$\{\}$	0000	−	−	−
$\{0\}$	0011	+	−	−
$\{1\}$	0110	−	+	−
$\{7\}$	1100	−	−	+
$\{0, 1\}$	0111	+	+	−
$\{0, 7\}$	1111	+	+	+
$\{1, 7\}$	1110	−	+	+
$\{0, 1, 7\}$	1111	+	+	+

Applications: Key-Value Storage Systems

- One application of Bloom Filters are key-value storage systems.
- These systems use *slow* secondary media to store key values.
- Not all candidate keys correspond to values.
- For simplicity, let's assume there's only one slow disk.
- We want to know if some value (and if so which) has key k.
- The query B`.contains(` k `)` may result in three cases:
  1. B returns `false`:
     - Avoids disk access.
  2. B returns `true` and $k \in B$.
     - Looking up the value using a slow method, returns the value.
  3. B returns `true` and $k \notin B$.
     - Looking up the value fails, we return $\perp$.
- If there are enough queries for bogus keys then this saves time.

Application: Detecting Malicious Websites

- Let's assume we want to know whether a given URL is malicious.
- We hash the URL into hash codes $h_0, \ldots, h_{n-1}$.
- If Bit $h_i = 0$ for some i s.t. $0 \leq i < n$, then the URL isn't know.
- Otherwise the URL is "known" (but this may be a false positive).
- There are two possibilities:
  - We trust the answer.
  - We use a slow database operation to see if the URL is known.

Probabilistic Spell Checkers

- We start with an empty Bloom Filter, B.
- For each allowed word, we add the word's hash code to B.
- The user enters a word.
- We compute the hash codes of the word.
- If some hash code bits aren't in the filter, the word is invalid.
- Otherwise, we assume the word is valid.

Pre-Processing Database Joins

- Let's assume we have two database tables T_1 and T_2.
- Let's compute the join $T_1 \bowtie T_2$.
- This is expensive and takes $\mathcal{O}(|T_1| \times |T_2|)$ worst-case time.
- Removing redundant rows from T_1 and T_2 takes much time.
- This also takes $\mathcal{O}(|T_1| \times |T_2|)$ worst-case time.
- Using Bloom Filters we do this in time $\mathcal{O}(|T_1| + |T_2|)$.

Pre-Processing Database Joins (Continued)

- Let's remove the redundant rows of T_1.
- Let S be the intersection of the scopes of T_1 and T_1.
- We start with an empty filter B.
- There are two phases:
  1. Add hash values of the rows in T_2 to B.
  2. Remove rows from T_1 whose hash values aren't in B.
- In practice, this pre-processing speeds up the join.

Analysis

- Time to decide membership is independent of current “size.”
  - With k constant-time hash functions we need $\mathcal{O}(k)$ time.
- Reliability may be improved by increasing the number of bits.
- Let R be the *false positive rate*.
- For ± 4.8 extra bits per member, R reduces by a factor of ± 10.

Question Time

Questions Anybody?

For Monday

- Study the presentation,
- Implement the bitmap class from scratch.

Summary

- We've studied n-bit bitmaps and m-bit Bloom Filters.
- Bitmap decides which numbers in $\{0, \ldots, n-1\}$ are in the set.
- The i-th member is in the set if and only if the ith bit is set.
- They are 100% accurate.
- With perfect hashing you can use them for other kinds of sets.
- Bloom Filters are probabilistic data structures.
- They use several hash functions, $h_i(\cdot)$.
- Candidate, c, isn't in the filter if Bit $h_i(c)$ is 0, for some i.
- Otherwise, Bit $h_i(c)$ is 1 for each i.
- In this case u is in the set, but false positives may occur.

About this Document

- This document was created with `pdflatex`.
- The LaTeX document class is `beamer`.