Detecting and Measuring Similarity in Code Clones

Randy Smith
Susan Horwitz

Department of Computer Sciences
University of Wisconsin—Madison

Code Clones

- Code clones a fact of life
  - Valuable for maintenance, refactoring, code compaction, debugging, plagiarism detection, etc.

- But, what is a clone?
  - Identical sequences of code?
  - Identical up to parameter substitution?
  - Similar sequences of code?

A Clone?

```
{
   const unsigned char *p1 = s1;
   const unsigned char *p2 = s2;
   unsigned char c1, c2;

   if (p1 == p2)
      return 0;

   do
   {
      c1 = TOLOWER(*p1++);
      c2 = TOLOWER(*p2++);
      if (c1 == '\0')
        break;
   }
   while (c1 == c2);

   return c1 - c2;
}
```

```
{
   const unsigned char *r1 = s1;
   const unsigned char *r2 = s2;
   unsigned char c1, c2;

   if (r1 == r2)
      return 0;

   do
   {
      c1 = TOLOWER(*r1++);
      c2 = TOLOWER(*r2++);
      if (c1 == '\0')
        break;
   }
   while (c1 == c2);

   return c1 - c2;
}
```

A Clone?

```c
{
   const unsigned char *p1 = s1;
   const unsigned char *p2 = s2;
   unsigned char c1, c2;

   if (p1 == p2)
      return 0;

   do
   {
      c1 = TOLOWER(*p1++);
      c2 = TOLOWER(*p2++);
      if (c1 == '\0')
        break;
   }
   while (c1 == c2);

   return c1 - c2;
}
```

```c
{
   const unsigned char *p1 = s1;
   const unsigned char *p2 = s2;
   unsigned char c1, c2;

   if (p1 == p2 || n == 0)
      return 0;

   do
   {
      c1 = TOLOWER(*p1++);
      c2 = TOLOWER(*p2++);
      if (c1 == '\0' || c1 != c2)
        return c1 - c2;
   }
   while (--n > 0);

   return c1 - c2;
}
```

A Clone?

```
{
   const unsigned char *p1 = s1;
   const unsigned char *p2 = s2;
   unsigned char c1, c2;

   if (p1 == p2)
      return 0;

   do
   {
      c1 = TOLOWER(*p1++);
      c2 = TOLOWER(*p2++);
      if (c1 == '\0')
        break;
   }
   while (c1 == c2);

   return c1 - c2;
}
```

```
{


   unsigned char c1, c2;

   if (*s1 == *s2 || n == 0)
      return 0;

   do
   {
      c1 = TOLOWER(*s1++);
      c2 = TOLOWER(*s2++);
      if (c1 == '\0' || c1 != c2)
        return c1 - c2;
      eq_cnt++;
   }
   while (--n > 0);

   full_end = 1;

   return c1 - c2;
}
```

Defining Clones

- Clone definition depends on its use

```
do {
  c1 = TOLOWER(*p1++);
  c2 = TOLOWER(*p2++);
  if (c1 == '\0')
    break;
} while (c1 == c2);
```

Defining Clones

- Clone definition depends on its use

```
do {
  c1 = TOLOWER(*p1++);
  c2 = TOLOWER(*p2++);
  if (c1 == '\0')
    break;
} while (c1 == c2);
```

Identity Similarity

```
do {
  c1 = TOLOWER(*p1++);
  c2 = TOLOWER(*p2++);
  if (c1 == '\0')
    break;
} while (c1 == c2);
```

Defining Clones

- Clone definition depends on its use

```
do {
  c1 = TOLOWER(*p1++);
  c2 = TOLOWER(*p2++);
  if (c1 == '\0')
    break;
} while (c1 == c2);
```

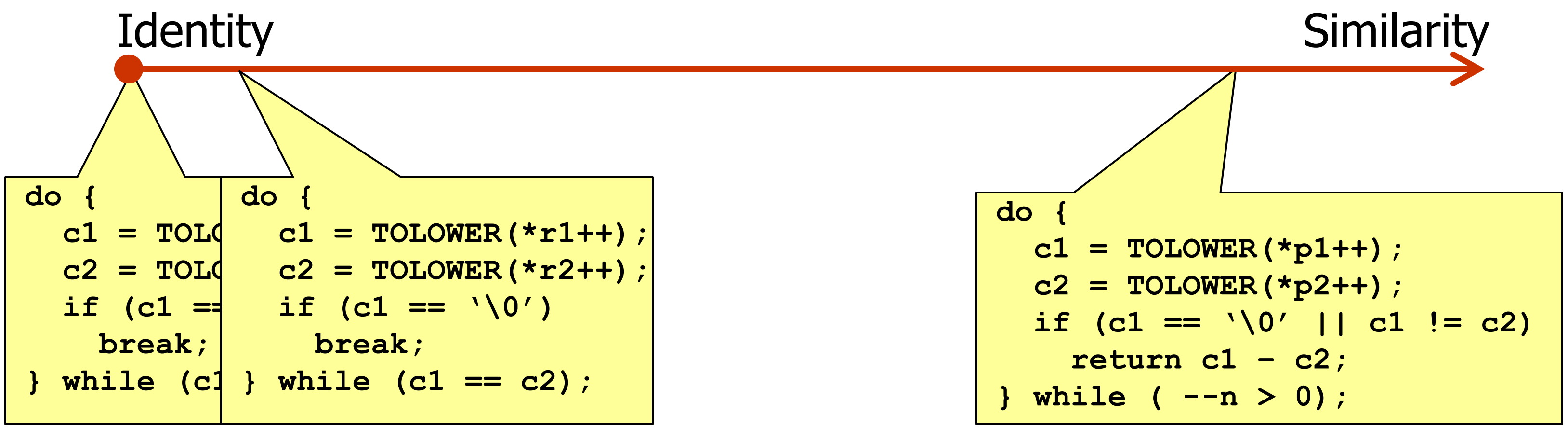

```
do {
  c1 = TOLO
  c2 = TOLO
  if (c1 ==
    break;
} while (c1
```

```
do {
  c1 = TOLOWER(*r1++);
  c2 = TOLOWER(*r2++);
  if (c1 == '\0')
    break;
} while (c1 == c2);
```

Defining Clones

- Clone definition depends on its use

```
do {
  c1 = TOLOWER(*p1++);
  c2 = TOLOWER(*p2++);
  if (c1 == '\0')
    break;
} while (c1 == c2);
```



- Position: similarity should be intrinsic to definition

This work

- We propose techniques for assessing and quantifying clone similarity.

- Contributions:
  - Clone detector with tunable degrees of similarity
  - *Similarity Score* and *Similarity Distance* for quantifying similarity degree
  - “similarity-aware” statement fingerprints

Outline

- Introduction and Position
- Quantifying Similarity
- Applications
- Conclusion

Quantifying Similarity: Statement Fingerprints

- Source code statements are atomic units
  - Abstract away identifier names, constant values, etc.
  - Statements are reconstructed sequences of tokens

- Goal: map statements that are similar but not necessarily identical to the same fingerprint

Quantifying Similarity: Statement Fingerprints

- Use n-grams

```
if ( id < int_lit || id ( id , id ) != int_lit ) break ;
```

Quantifying Similarity: Statement Fingerprints

- Use n-grams

```
if ( id < int_lit || id ( id , id ) != int_lit ) break ;
```

```
if ( id <
```

Quantifying Similarity: Statement Fingerprints

- Use n-grams

```
if ( id < int_lit || id ( id , id ) != int_lit ) break ;
```

```
if ( id <
( id < int_lit
```

Quantifying Similarity: Statement Fingerprints

- Use n-grams

```
if ( id < int_lit || id ( id , id ) != int_lit ) break ;
```

```
if ( id <
( id < int_lit
id < int_lit ||
```

Quantifying Similarity: Statement Fingerprints

- Use n-grams

```
if ( id < int_lit || id ( id , id ) != int_lit ) break ;
```

```
if ( id <
( id < int_lit
id < int_lit ||
< int_lit || id
```

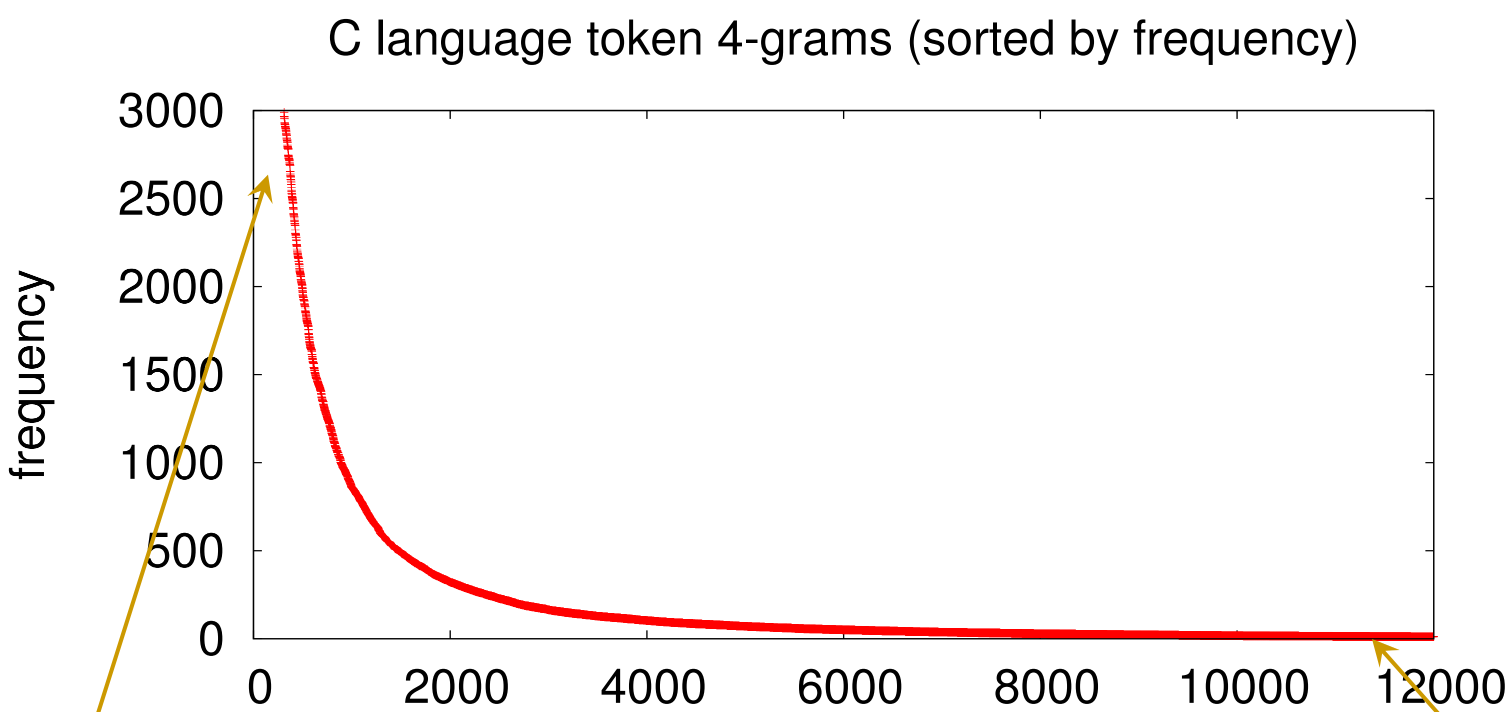
Quantifying Similarity: Statement Fingerprints

- Use n-grams

```
if ( id < int_lit || id ( id , id ) != int_lit ) break ;
```

```
if ( id <                         id ) != int_lit
( id < int_lit                    ) != int_lit )
id < int_lit ||        . . .      != int_lit ) break
< int_lit || id                   int_lit ) break ;
```

Similarity-preserving Fingerprints




```
int_lit , int_lit ,
, int_lit , int_lit
id ( id ,
id ( id )
```

```
float_lit ) return -
int ) ] )
int unsigned id ;
sizeof id [ id
```

Computing Fingerprints

Compute_Fingerprint(Statement S, int n, int k):

1. for each n-gram s $\in$ S, lookup *frequency* (s)
2. select the least-frequent k n-grams in S
3. concatenate k n-grams in occurrence order in S
4. return fingerprint

Computing Fingerprints

```
if ( id < int_lit || id ( id , id ) != int_lit ) break ;
```

<table>
<tr><th>4-gram</th><th>Freq.</th></tr>
<tr><td>if (id <</td><td>8614</td></tr>
<tr><td>(id < int_lit</td><td>6988</td></tr>
<tr><td>id < int_lit ||</td><td>984</td></tr>
<tr><td>< int_lit || id</td><td>1008</td></tr>
<tr><td>int_lit || id (</td><td>420</td></tr>
<tr><td>|| id (id</td><td>3050</td></tr>
<tr><td>id (id ,</td><td>117967</td></tr>
<tr><td>(id , id</td><td>77921</td></tr>
<tr><td>id , id)</td><td>64951</td></tr>
<tr><td>, id) !=</td><td>532</td></tr>
<tr><td>id) != int_lit</td><td>1111</td></tr>
<tr><td>) != int_lit)</td><td>1722</td></tr>
<tr><td>!= int_lit) break</td><td>102</td></tr>
<tr><td>int_lit) break ;</td><td>734</td></tr>
</table>

```
int_lit || id (    , id ) !=    != int_lit ) break

45:10:31:08        02:31:09:27  27:45:09:16

45:10:31:08:02:31:09:27:27:45:09:16
```

General parameters:

- n: width of an n-gram
- k: number of n-grams

Fingerprint Justification

- Three facets to the fingerprint function:

1. Using least frequent n-grams…
   …captures presence of uncommon tokens

2. N-grams vs. individual tokens…
   …reflects importance of token ordering

3. Using just *k* n-grams means…
   …distinct statements can have same fingerprint

Similar Statements

1. Two for-loops that differ in initialization

```
for (id = int_lit; id[id] && id( id[id] ); id++)
for (id = id; id[id] && id( id[id] ); id++)
```

2. Three assignments with differing parameters

```
id = id ( id->id->id , id );
id = id ( id->id->id , id , id );
id = id ( id->id->id , id , id , id );
```

3. Distinct consequent in conditional

```
for ( id = int_lit ; id < id ; id++ )
  if ( id [id] && (id == int_lit || (id && id ( id ))))
     id ( id ) ;

for ( id = int_lit ; id < id ; id++ )
  if ( id [id] && (id == int_lit || (id && id ( id ))))
     id ( id , int_lit ) ;
```

Measuring Similarity in Blocks

```
{
  const unsigned char *p1 = s1;
  const unsigned char *p2 = s2;
  unsigned char c1, c2;

  if (p1 == p2)
     return 0;

  do
  {
     c1 = TOLOWER(*p1++);
     c2 = TOLOWER(*p2++);
     if (c1 == '\0')
       break;
  }
  while (c1 == c2);

  return c1 - c2;
}
```

- Identify clones at the block level
- Code blocks
  - Def: set of statements (fingerprints) between matching braces
  - Smallest measurable unit

Similiarity Measures

- Similarity Score:

$$sim(S_1, S_2) = \left\langle \frac{|S_1 \cap S_2|}{|S_1|}, \frac{|S_1 \cap S_2|}{|S_2|} \right\rangle$$

- Measures fraction of each block common to both blocks, respectively
- Computes on *sets* of fingerprints – ordering not relevant

Similiarity Measures

- Similarity Distance:

$$sim\_dist(\langle s_1, s_2 \rangle) = 1 - \sqrt{\frac{(s_1)^2 + (s_2)^2}{2}}$$

- Converts similarity score to single value
  - (useful for rank ordering, etc.)
- Normalized to value between 0 and 1

A Clone?

```
{
   const unsigned char *p1 = s1;
   const unsigned char *p2 = s2;
   unsigned char c1, c2;

   if (p1 == p2)
      return 0;

   do
   {
      c1 = TOLOWER(*p1++);
      c2 = TOLOWER(*p2++);
      if (c1 == '\0')
        break;
   }
   while (c1 == c2);

   return c1 - c2;
}
```

```
{
   const unsigned char *r1 = s1;
   const unsigned char *r2 = s2;
   unsigned char c1, c2;

   if (r1 == r2)
      return 0;

   do
   {
      c1 = TOLOWER(*r1++);
      c2 = TOLOWER(*r2++);
      if (c1 == '\0')
        break;
   }
   while (c1 == c2);

   return c1 - c2;
}
```

A Clone?

sim(left,right) = < 10/10, 10/10> = <1, 1>

sim_dist(<1,1>) = 0

```
{
   const unsigned char *p1 = s1;
   const unsigned char *p2 = s2;
   unsigned char c1, c2;

   if (p1 == p2)
      return 0;

   do
   {
      c1 = TOLOWER(*p1++);
      c2 = TOLOWER(*p2++);
      if (c1 == '\0')
        break;
   }
   while (c1 == c2);

   return c1 - c2;
}
```

```
{
   const unsigned char *r1 = s1;
   const unsigned char *r2 = s2;
   unsigned char c1, c2;

   if (r1 == r2)
      return 0;

   do
   {
      c1 = TOLOWER(*r1++);
      c2 = TOLOWER(*r2++);
      if (c1 == '\0')
        break;
   }
   while (c1 == c2);

   return c1 - c2;
}
```

A Clone?

```
{
   const unsigned char *p1 = s1;
   const unsigned char *p2 = s2;
   unsigned char c1, c2;

   if (p1 == p2)
      return 0;

   do
   {
      c1 = TOLOWER(*p1++);
      c2 = TOLOWER(*p2++);
      if (c1 == '\0')
        break;
   }
   while (c1 == c2);

   return c1 - c2;
}
```

```
{
   const unsigned char *p1 = s1;
   const unsigned char *p2 = s2;
   unsigned char c1, c2;

   if (p1 == p2 || n == 0)
      return 0;

   do
   {
      c1 = TOLOWER(*p1++);
      c2 = TOLOWER(*p2++);
      if (c1 == '\0' || c1 != c2)
        return c1 - c2;
   }
   while (--n > 0);

   return c1 - c2;
}
```

A Clone?

sim(left,right) = < 7/10, 7/10> = <0.7, 0.7>

sim_dist(<0.7, 0.7>) = 0.3

```
{
   const unsigned char *p1 = s1;
   const unsigned char *p2 = s2;
   unsigned char c1, c2;

   if (p1 == p2)
      return 0;

   do
   {
      c1 = TOLOWER(*p1++);
      c2 = TOLOWER(*p2++);
      if (c1 == '\0')
        break;
   }
   while (c1 == c2);

   return c1 - c2;
}
```

```
{
   const unsigned char *p1 = s1;
   const unsigned char *p2 = s2;
   unsigned char c1, c2;

   if (p1 == p2 || n == 0)
      return 0;

   do
   {
      c1 = TOLOWER(*p1++);
      c2 = TOLOWER(*p2++);
      if (c1 == '\0' || c1 != c2)
        return c1 - c2;
   }
   while (--n > 0);

   return c1 - c2;
}
```

A Clone?

```
{
   const unsigned char *p1 = s1;
   const unsigned char *p2 = s2;
   unsigned char c1, c2;

   if (p1 == p2)
      return 0;

   do
   {
      c1 = TOLOWER(*p1++);
      c2 = TOLOWER(*p2++);
      if (c1 == '\0')
        break;
   }
   while (c1 == c2);

   return c1 - c2;
}
```

```
{


   unsigned char c1, c2;

   if (*s1 == *s2 || n == 0)
      return 0;

   do
   {
      c1 = TOLOWER(*s1++);
      c2 = TOLOWER(*s2++);
      if (c1 == '\0' || c1 != c2)
        return c1 - c2;
      eq_cnt++;
   }
   while (--n > 0);

   full_end = 1;

   return c1 - c2;
}
```

A Clone?

sim(left,right) = < 5/10, 5/10> = <0.5, 0.5>

sim_dist(<0.5, 0.5>) = 0.5

```
{
   const unsigned char *p1 = s1;
   const unsigned char *p2 = s2;
   unsigned char c1, c2;

   if (p1 == p2)
      return 0;

   do
   {
      c1 = TOLOWER(*p1++);
      c2 = TOLOWER(*p2++);
      if (c1 == '\0')
        break;
   }
   while (c1 == c2);

   return c1 - c2;
}
```

```
   unsigned char c1, c2;

   if (*s1 == *s2 || n == 0)
      return 0;

   do
   {
      c1 = TOLOWER(*s1++);
      c2 = TOLOWER(*s2++);
      if (c1 == '\0' || c1 != c2)
        return c1 - c2;
      eq_cnt++;
   }
   while (--n > 0);

   full_end = 1;

   return c1 - c2;
}
```

Application: Clusters of Clones

- Many blocks may be mutually similar, but similarity score (and clone pairing) is binary

- Use clique-based clustering to find clone groups
  - Compute Similarity Distance between each pair
  - Convert to graph:
    - Blocks are nodes

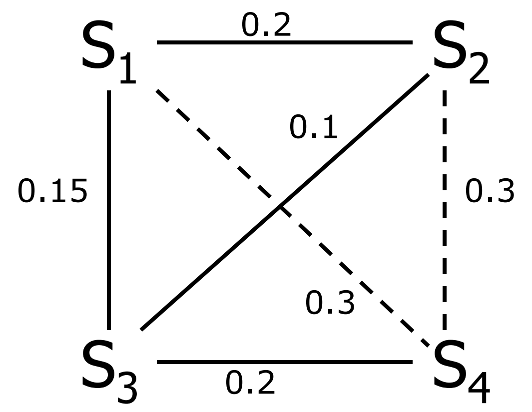
S_1 S_2

S_3 S_4

Application: Clusters of Clones

- Many blocks may be mutually similar, but similarity score (and clone pairing) is binary

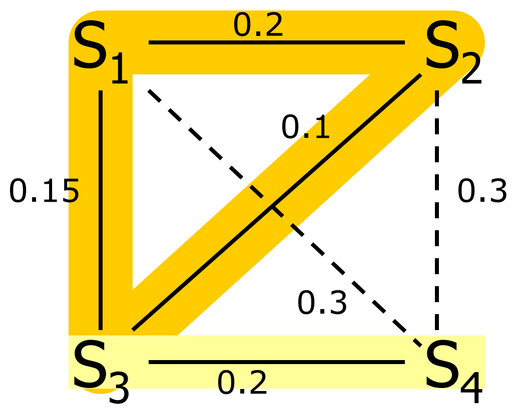
- Use clique-based clustering to find clone groups
  - Compute Similarity Distance between each pair
  - Convert to graph:
    - Blocks are nodes
    - Place edge if within threshold



Application: Clusters of Clones

- Many blocks may be mutually similar, but similarity score (and clone pairing) is binary

- Use clique-based clustering to find clone groups
  - Compute Similarity Distance between each pair
  - Convert to graph:
    - Blocks are nodes
    - Place edge if within threshold
  - Find maximal cliques
    - $\{S_1, S_2, S_3\}$, $\{S_3, S_4\}$



Application: Clusters of Clones

```
for ( ; alphanum(c); c =dgetc(dotc,dapc, out))
{
 if ( t < TOKLEN)
    token[t++] = c;
 else {
    token[t] = '\0';
    fprintf(stderr, "dappp:%s :%d: token too "
           "long: %s \n", dotname, lineno,
           token);
    exit(1) ;
 }
}
unget1c(c, dotc, (out ? dapc : NULL) ) ;
```

```
for (t = 0 ; c == '.' ||
             ( '0' <= c && c <= '9') ;
     c = sbsgetc(sbsfile))
{
 if ( t < TOKLEN)
    token[t++] = c;
 else {
    token[t] = '\0';
    fprintf(stderr, "sbstrans: before %d:"
             "token too long: %s\n",
             sbslineno, token);
    exit(1);
 }
}
```

```
for ( ; num(c); c =dgetc(dotc,dapc, out))
{
 if ( t < TOKLEN)
    token[t++] = c;
 else {
    token[t] = '\0';
    fprintf(stderr, "dappp:%s :%d: token too"
           "long: %s \n", dotname, lineno,
           token);
    exit(1) ;
 }
}
unget1c(c, dotc, (out ? dapc : NULL) ) ;
```

```
for (t=0; ('a' <= c && c <= 'z') ||
          ('A' <= c && c <= 'Z') ||
          ('0' <= c && c <= '9') ||
          c == ' ' || c == '.';
     c = sbsgetc(sbsfile))
{
   if (t < TOKENLEN)
      token [t++] = c;
   else {
      token[t] = '\0';
      fprintf(stderr, "sbstrans: before %d:"
              "token too long: %s\n",
       sbslineno, token);
      exit(1);
   }
}
```

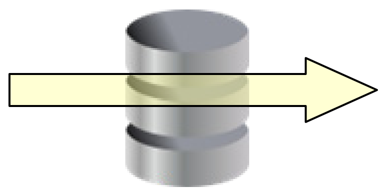
Application: Rank Ordering

- Given block S, find all blocks similar to it

- Can rank-order blocks by similarity distance
  - Inverted index keyed on fingerprint value restricts computation to blocks with overlap

```
for ( ; alphanum(c);
        c =dgetc(dotc,dapc,out))
{
  if ( t < TOKLEN)
      token[t++] = c;
  else {
      token[t] = '\0';
      fprintf(stderr, "dappp:%s :%d: "
          "token too long: %s \n",
          dotname, lineno, token);
      exit(1) ;
  }
}
ungetlc(c, dotc, (out ? dapc : NULL));
```

S



clone repository

```
sim< S, base:01:00 > = <1.00, 1.00> dst=0.000
sim< S, base:01:01 > = <0.67, 0.67> dst=0.333
sim< S, base:01:03 > = <0.50, 0.60> dst=0.448
sim< S, base:01:02 > = <0.50, 0.50> dst=0.500
.              .              .         .
.              .              .         .
.              .              .         .
```

Conclusion

- Similarity should be intrinsic to clone definition

- Two levels of similarity:
  - statement fingerprints tend to preserve similarity
  - Similarity Score quantifies similarity for blocks

- Quantified similarity enables interesting applications

Detecting and Measuring Similarity in Code Clones

Thank you