UNIVERSIDAD NACIONAL DE COLOMBIA

Estructuras de Datos

Sesión 8

Queue Applications

Yoan Pinzón

Table of Content Session 8

- **Queue Applications**
  - ▷ Image-Component Labeling
  - ▷ Lee's Wire Router

Queue Application

Image-Component Labeling

		1				
		1	1			
				1		
			1	1		
	1			1		1
1	1	1				1
1	1	1			1	1

(a) Input

		2				
		2	2			
				3		
			3	3		
	4			3		5
4	4	4				5
4	4	4			5	5

(b) Output

- Digitised image: $m \times m$ matrix of pixels (0,1).
  0-pixel represents image background;
  1-pixel represents a point on an image component.

- Two pixels are adjacent if one is to the left, above, right, or below the other.

- Two 1-pixels (component pixels) that are adjacent belong to the same image component.

Objective: Label the component pixels such that two pixels get the same label if and only if they are pixels of the same image component.

File ImageComponents.java

```java
package unal.applications;

import unal.datastructures.*;
import java.util.*;

public class ImageComponents
{
   // fields
   static int[][] pixel;
   static int size; // number of rows and columns in the image

   // methods
   /** input the image */
   private static void inputImage( )
   {
      // define the input stream to be the standard input stream
      Scanner s = new Scanner( System.in );

      System.out.println( "Enter␣image␣size" );
```

```java
      size = s.nextInt( );

      // create and input the pixel array
      pixel = new int[ size + 2 ][ size + 2 ];
      System.out.println( "Enter␣the␣pixel␣array␣in␣row-major␣↙
         ↪ order" );
      for( int i = 1; i <= size; i++ )
         for( int j = 1; j <= size; j++ )
            pixel[ i ][ j ] = s.nextInt( );
   }

   /** label the components */
   private static void labelComponents( )
   {
      // initialize offsets
      Position[] offset = new Position[ 4 ];
      offset[ 0 ] = new Position( 0, 1 ); // right
      offset[ 1 ] = new Position( 1, 0 ); // down
      offset[ 2 ] = new Position( 0, -1 ); // left
      offset[ 3 ] = new Position( -1, 0 ); // up

      // initialize wall of 0 pixels
```

```
  for( int i = 0; i <= size + 1; i++ )
  {
    pixel[ 0 ][ i ] = pixel[ size + 1 ][ i ] = 0; // bottom and top
    pixel[ i ][ 0 ] = pixel[ i ][ size + 1 ] = 0; // left and right
  }

  int numOfNbrs = 4; // neighbors of a pixel position
  ArrayQueue<Position> q = new ArrayQueue<>( );
  Position nbr = new Position( );
  int id = 1; // component id

  // scan all pixels labeling components
  for( int r = 1; r <= size; r++ )   // row r of image
    for( int c = 1; c <= size; c++ ) // column c of image
     if( pixel[ r ][ c ] == 1 )
     {  // new component
       pixel[ r ][ c ] = ++id; // get next id
       Position here = new Position( r, c );
       do
       {  // find rest of component
```

```
         for( int i = 0; i < numOfNbrs; i++ )
         {  // check all neighbors of here
           nbr.row = here.row + offset[ i ].row;
           nbr.col = here.col + offset[ i ].col;
           if( pixel[ nbr.row ][ nbr.col ] == 1 )
           {  // pixel is part of current component
             pixel[ nbr.row ][ nbr.col ] = id;
             q.put( new Position( nbr.row, nbr.col ) );
           }
         }
         // any unexplored pixels in component?
         here = q.remove( ); // a component pixel
       } while( here != null );
     } // end of if, for c, and for r
}

/** output labeled image */
private static void outputImage( )
{
  System.out.println( "The␣labeled␣image␣is" );
  for( int i = 1; i <= size; i++ )
  {
```

```java
         for( int j = 1; j <= size; j++ )
            System.out.print( pixel[ i ][ j ] + "␣␣" );
         System.out.println( );
      }
   }

   /** entry point for component labeling program */
   public static void main ( String[] args )
   {
      inputImage( );
      labelComponents( );
      outputImage( );
   }
}

class Position
{
   // fields
   int row; // row number of the position
   int col; // column number of the position

   // constructors
```

```java
   Position( )
   {
      this( 0, 0 );
   }

   Position( int row, int col )
   {
      this.row = row;
      this.col = col;
   }

   // convert to string suitable for output
   @Override
   public String toString( )
   {
      return new String( row + "␣" + col );
   }
}
```

File `ImageComponents.input`

```
7
0 0 1 0 0 0 0
0 0 1 1 0 0 0
0 0 0 0 1 0 0
0 0 0 1 1 0 0
1 0 0 0 1 0 0
1 1 1 0 0 0 0
1 1 1 0 0 0 0
```

Compiling `ImageComponents.java`

```
C:\2016699\code> javac unal\applications\ImageComponents.java ↵
C:\2016699\code> java unal.applications.ImageComponents < unal\app
lications\ImageComponents.input > unal\applications\ImageComponent
s.output ↵
```

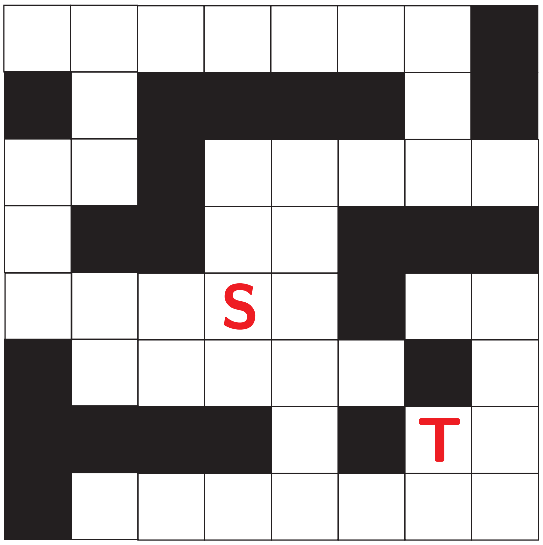
File `ImageComponents.output`

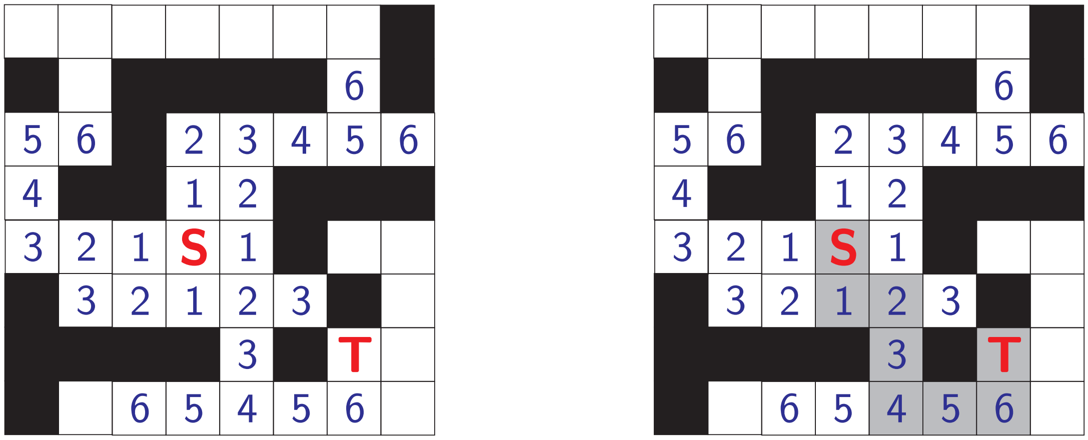
```
Enter image size
Enter the pixel array in row-major order
The labeled image is
0  0  2  0  0  0  0
0  0  2  2  0  0  0
0  0  0  0  3  0  0
0  0  0  3  3  0  0
4  0  0  0  3  0  0
4  4  4  0  0  0  0
4  4  4  0  0  0  0
```

Queue Application

Lee's Wire Router

Find a path from **S** to **T** by *wave propagation*.





(a) Filing

(b) Retrace

File WireRouter.java

```
package unal.applications;

import unal.datastructures.*;
import java.util.*;

public class WireRouter
{
   // fields
   static int[][] grid;
   static int size;            // number of rows and columns in the grid
   static Position start, finish; // both end points of wire
   static Position[] path;     // the shortest path

   // methods
   /** input the wire routing data */
   private static void inputData( )
   {
      // define the input stream to be the standard input stream
      Scanner s = new Scanner( System.in );

      System.out.println( "Enter␣grid␣size" );
      size = s.nextInt( );

      System.out.println( "Enter␣the␣start␣position" );
      start = new Position( s.nextInt( ), s.nextInt( ) );
      System.out.println( "Enter␣the␣finish␣position" );
      finish = new Position( s.nextInt( ), s.nextInt( ) );

      // create and input the wiring grid array
      grid = new int [ size + 2 ][ size + 2 ];
      System.out.println( "Enter␣the␣wiring␣grid␣in␣row-major␣↙
        ↳ order" );
      for( int i = 1; i <= size; i++ )
         for( int j = 1; j <= size; j++ )
            grid[ i ][ j ] = s.nextInt( );
   }

   /** find a shortest path from start to finish
    * @return true if successful, false if impossible */
   private static boolean findPath( )
```

```
   {
      if( ( start.row == finish.row ) && ( start.col == finish.col ↙
        ↳ ) )
         return true;

      // initialize offsets
      Position[] offset = new Position[ 4 ];
      offset[ 0 ] = new Position( 0, 1 ); // right
      offset[ 1 ] = new Position( 1, 0 ); // down
      offset[ 2 ] = new Position( 0, -1 ); // left
      offset[ 3 ] = new Position( -1, 0 ); // up

      // initialize wall of blocks around the grid
      for( int i = 0; i <= size + 1; i++ )
      {
         grid[ 0 ][ i ] = grid[ size + 1 ][ i ] = 1; // bottom and top
         grid[ i ][ 0 ] = grid[ i ][ size + 1 ] = 1; // left and right
      }

      Position here = new Position( start.row, start.col );
      grid[ start.row ][ start.col ] = 2; // block
      int numOfNbrs = 4; // neighbors of a grid position
      // label reachable grid positions
      ArrayQueue<Position> q = new ArrayQueue<>( );
      Position nbr = new Position( );
      do
      {  // label neighbors of here
         for( int i = 0; i < numOfNbrs; i++ )
         {  // check out neighbors of here
            nbr.row = here.row + offset[ i ].row;
            nbr.col = here.col + offset[ i ].col;
            if( grid[ nbr.row ][ nbr.col ] == 0 )
            {  // unlabeled nbr, label it
               grid[ nbr.row ][ nbr.col ] = grid[ here.row ][ ↙
                 ↪ here.col ] + 1;
               if( ( nbr.row == finish.row ) && ( nbr.col == ↙
                 ↪ finish.col ) )
                  break; // done
               // put on queue for later expansion
               q.put( new Position( nbr.row, nbr.col ) );
            }
         }
```

```
         // have we reached finish?
         if( ( nbr.row == finish.row ) && ( nbr.col == finish.col ) )
            break; // done

         // finish not reached, can we move to a nbr?
         if( q.isEmpty( ) ) return false;      // no path
         here = q.remove( ); // get next position
      } while( true );

      // construct path
      int pathLength = grid[ finish.row ][ finish.col ] - 2;
      path = new Position[ pathLength ];

      // trace backwards from finish
      here = finish;
      for( int j = pathLength - 1; j >= 0; j-- )
      {
         path[ j ] = here;
         // find predecessor position
         for( int i = 0; i < numOfNbrs; i++ )
         {
            nbr.row = here.row + offset[ i ].row;
            nbr.col = here.col + offset[ i ].col;
            if( grid[ nbr.row ][ nbr.col ] == j + 2 ) break;
         }
         here = new Position( nbr.row, nbr.col ); // move to predecessor
      }

      return true;
   }

   /** output path to exit */
   private static void outputPath( )
   {
      System.out.println( "The␣wire␣path␣is" );
      for( Position x : path )
         System.out.println( x );
   }

   /** entry point for wire routing program */
   public static void main( String[] args )
   {
      inputData( );
```

```
      if( findPath( ) ) outputPath( );
      else System.out.println( "There␣is␣no␣wire␣path" );
   }
}

class Position
{
   // fields
   int row; // row number of the position
   int col; // column number of the position

   // constructors
   Position( )
   {
      this( 0, 0 );
   }

   Position( int row, int col )
   {
      this.row = row;
      this.col = col;
   }
```

```
    // convert to string suitable for output
    @Override
    public String toString( )
    {
        return new String( row + "␣" + col );
    }
}
```

File `WireRouter.input`

```
7
3 2
4 6
0 0 1 0 0 0 0
0 0 1 1 0 0 0
0 0 0 0 1 0 0
0 0 0 1 1 0 0
1 0 0 0 1 0 0
1 1 1 0 0 0 0
1 1 1 0 0 0 0
```

Compiling WireRouter.java

```
C:\2016699\code> javac unal\applications\WireRouter.java ↵
C:\2016699\code> java unal.applications.WireRouter < unal\applica
tions\WireRouter.input > unal\applications\WireRouter.output ↵
```

File WireRouter.output

```
Enter grid size
Enter the start position
Enter the finish position
Enter the wiring grid in row-major order
The wire path is
4 2
5 2
5 3
5 4
6 4
6 5
6 6
5 6
4 6
```