VT ACM ICPC Handbook

ACM Programming Team 2012

September 20, 2012

Contents

Preface

This book is intended as a reference, to be used both during the competition as well in preparation for it.

It is hosted on github at https://github.com/VTACMProgrammingTeam/ICPCHandbook. If you wish to contribute, please send email to godmar@gmail.com

Chapter 1

String Processing

1.1 Regular Expressions

1.1.1 Regular Expressions

1.1.2 Zero-width Lookahead Technique

1.1.3 NFA Simulation

The Regex engine in Java does not convert to a Thompson-DFA; it uses a backtracking algorithm to find out if a regular expression matches a string. This leads to pathological cases with exponential runtime increase, particularly when the regular expression contains a large number of Kleene stars.

In those situations, it may be helpful to construct your own mini-regexp interpreter by building and simulating an NFA (nondeterministic finite automaton).

Example problem is NCPC 2011/E where the input are globs such as `*a*a*a*a` that should be matched against filenames.

```
import sys, string
from collections import defaultdict

def NextLine(): return sys.stdin.readline().rstrip()

class State:
    def __init__(self):
```

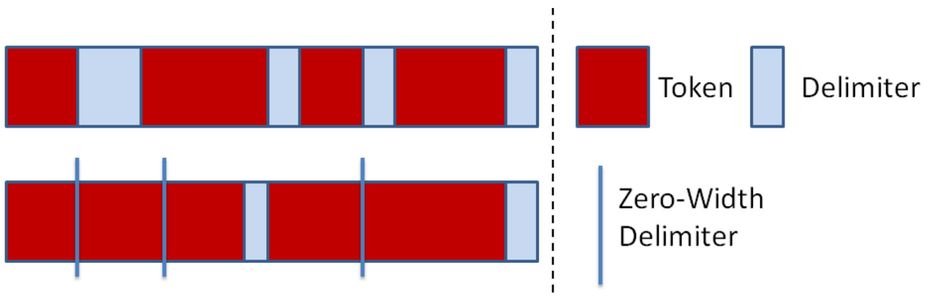


Figure 1.1: Lookahead Splitting

```
        self.transitions = defaultdict(set)

    def addTransition(self, symbol, destState):
        self.transitions[symbol].add(destState)

pattern = NextLine()
firststate = laststate = State()

for p in pattern:
    if p == '*':
        # wildcard - add self transitions for all
        # input characters and .
        for l in string.lowercase + '.':
            laststate.addTransition(l, laststate)
    else:
        # add transition to next state
        nextstate = State()
        laststate.addTransition(p, nextstate)
        laststate = nextstate

# lastState is now goal state for a full match

N = int(NextLine())
for i in range(N):
    fname = NextLine()
    # simulate NFA
    activestates = set([firststate])
    for f in fname:
        activestates = set(d for s in activestates \
                                 for d in s.transitions[f])

    if laststate in activestates:
        print fname
```

1.2 Parsing

1.2.1 Recursive Descent

Chapter 2

Geometry

2.1 Basics

A determinant of a $2x2$ matrix is defined as

$$\begin{vmatrix} a & b \\ c & d \end{vmatrix} = ad - bc$$

2.2 java.awt.geom

The `java.awt.geom` and `java.awt` packages have, albeit limited, facilities for geometric problems. There are classes to represent shapes - see java.awt.Shape, including lines, ellipses, rectangles and some curves.

- "is contained in". java.awt.geom.Shape provide a contains() method to test if a point is contained in a shape. Contains() returns true if the point is in the interior, and false if the point is outside the shape. However, it may return true or false if the point is on the shape boundary.
- "intersects." Tests if a shape intersects with a rectangle. Can also test if two lines or line segments intersect, but cannot find the point of intersection.

2.3 Coordinate Geometry

2.3.1 Line/Line Intersection

$$P_x = \frac{\begin{vmatrix} \begin{vmatrix} x_1 & y_1 \\ x_2 & y_2 \end{vmatrix} & \begin{vmatrix} x_1 & 1 \\ x_2 & 1 \end{vmatrix} \\ \begin{vmatrix} x_3 & y_3 \\ x_4 & y_4 \end{vmatrix} & \begin{vmatrix} x_3 & 1 \\ x_4 & 1 \end{vmatrix} \end{vmatrix}}{\begin{vmatrix} \begin{vmatrix} x_1 & 1 \\ x_2 & 1 \end{vmatrix} & \begin{vmatrix} y_1 & 1 \\ y_2 & 1 \end{vmatrix} \\ \begin{vmatrix} x_3 & 1 \\ x_4 & 1 \end{vmatrix} & \begin{vmatrix} y_3 & 1 \\ y_4 & 1 \end{vmatrix} \end{vmatrix}} \quad P_y = \frac{\begin{vmatrix} \begin{vmatrix} x_1 & y_1 \\ x_2 & y_2 \end{vmatrix} & \begin{vmatrix} y_1 & 1 \\ y_2 & 1 \end{vmatrix} \\ \begin{vmatrix} x_3 & y_3 \\ x_4 & y_4 \end{vmatrix} & \begin{vmatrix} y_3 & 1 \\ y_4 & 1 \end{vmatrix} \end{vmatrix}}{\begin{vmatrix} \begin{vmatrix} x_1 & 1 \\ x_2 & 1 \end{vmatrix} & \begin{vmatrix} y_1 & 1 \\ y_2 & 1 \end{vmatrix} \\ \begin{vmatrix} x_3 & 1 \\ x_4 & 1 \end{vmatrix} & \begin{vmatrix} y_3 & 1 \\ y_4 & 1 \end{vmatrix} \end{vmatrix}}$$

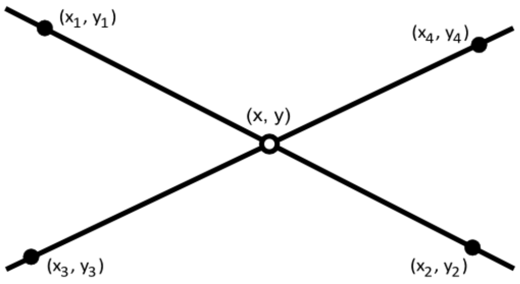


Figure 2.1: Line line intersection

The determinants can be written out as:

$$(P_x, P_y) = \left(\frac{(x_1y_2 - y_1x_2)(x_3 - x_4) - (x_1 - x_2)(x_3y_4 - y_3x_4)}{(x_1 - x_2)(y_3 - y_4) - (y_1 - y_2)(x_3 - x_4)}, \frac{(x_1y_2 - y_1x_2)(y_3 - y_4) - (y_1 - y_2)(x_3y_4 - y_3x_4)}{(x_1 - x_2)(y_3 - y_4) - (y_1 - y_2)(x_3 - x_4)} \right)$$

Source: http://en.wikipedia.org/wiki/Line-line_intersection.

Notes

- Parallel or coincident lines: Denominator will be zero:

$$(x_1 - x_2)(y_3 - y_4) - (y_1 - y_2)(x_3 - x_4) = 0$$

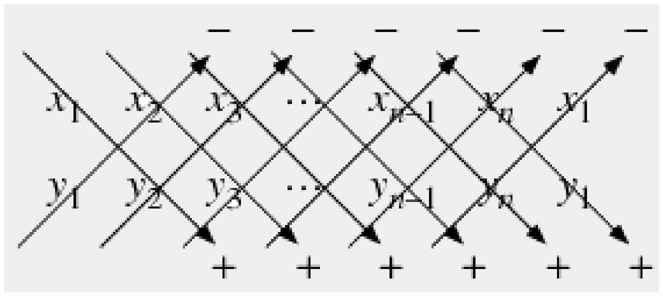
- One line horizontal ($y_1 = y_2$ or $y_3 = y_4$), the other vertical ($x_1 = x_2$ or $x_3 = x_4$) will also give a 0 determinant, but the lines will intersect. Handle as special case if problem allows it.
- Intersection point may be outside the given segments.
- If you only need to know if two lines intersect, but not where, use java.awt.geom.Line2D.intersects.

Code

```
static double det(double x1, double y1, double x2, double y2) {
    return x1 * y2 - y1 * x2;
}

static Point2D.Double intersects(Point2D.Double p1, Point2D.Double p2,
                                 Point2D.Double p3, Point2D.Double p4) {
    double d = det(p1.x - p2.x, p1.y - p2.y, p3.x - p4.x, p3.y - p4.y);
    double x12 = det(p1.x, p1.y, p2.x, p2.y);
    double x34 = det(p3.x, p3.y, p4.x, p4.y);

    // assert d != 0 (lines are known to intersect)
    double x = det(x12, p1.x-p2.x, x34, p3.x-p4.x) / d;
    double y = det(x12, p1.y-p2.y, x34, p3.y-p4.y) / d;
    return new Point2D.Double(x, y);
}
```



x_1 x_2 x_3 ... x_{n-1} x_n x_1
y_1 y_2 y_3 ... y_{n-1} y_n y_1
− − − − − −
+ + + + + +

Figure 2.2: Line line intersection

2.3.2 Area of a Polygon

The signed area of a planar non-self-intersecting polygon with vertices $(x_1, y_1), \ldots, (x_n, y_n)$ is

$$A = \frac{1}{2}\left(\begin{vmatrix} x_1 & x_2 \\ y_1 & y_2 \end{vmatrix} + \begin{vmatrix} x_2 & x_3 \\ y_2 & y_3 \end{vmatrix} + \ldots + \begin{vmatrix} x_n & x_1 \\ y_n & y_1 \end{vmatrix} \right)$$

Figure 2.2 shows how to multiply this out

$$A = \frac{1}{2}(x_1y_2 - x_2y_1 + x_2y_3 - x_3y_2 + \ldots + +x_{n-1}y_n - x_ny_{n-1} + x_ny_1 - x_1y_n)$$

(Source: Mathworld [1])

Notes

- Works for any simple polygon (concave or convex)
- Does not work for complex polygons (when any edges intersect)
- A is positive if points are in counterclockwise order, negative if points are in clockwise order. See the use Math.abs() in code below.
- Triangle and any Quadrilateral are, of course, just special cases.

Code

```
static double areaPolygon(Point2D.Double p[]) {
    double area = 0.0;
    for (int i = 0; i < p.length; i++) {
        area += p[i].x * p[(i+1) % p.length].y - p[i].y * p[(i+1) % p.length].x;
    }
    return Math.abs(area/2.0);
}
```

Bibliography

[1] Eric W. Weisstein. Polygon area. From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/PolygonArea.html.

Index