VT ACM ICPC Handbook

ACM Programming Team 2012

September 22, 2012

Contents

Preface v

1 Standard Libraries 1
- 1.1 Collection Classes 1
  - 1.1.1 Object Equivalence 1
  - 1.1.2 Comparators 2

2 Simulation 3
- 2.1 Monte Carlo Methods 3
- 2.2 Discrete Event Simulation 3

3 Searching 7
- 3.1 Backtracking 7
- 3.2 State Space Exploration 7
  - 3.2.1 Breadth-First Search (BFS) 7
  - 3.2.2 A^* 9

4 String Processing 11
- 4.1 Regular Expressions 11
  - 4.1.1 Regular Expressions 11
  - 4.1.2 Zero-width Lookahead Technique 11
  - 4.1.3 NFA Simulation 13
- 4.2 Parsing 15
  - 4.2.1 Recursive Descent 15

5 Geometry 17
- 5.1 Basics 17
- 5.2 java.awt.geom 17
- 5.3 Coordinate Geometry 17
  - 5.3.1 Line/Line Intersection 17
  - 5.3.2 Area of a Polygon 19

6 Gotchas 21

7 Mid-Atlantic Problem Sets 23
- 7.1 2005 23
  - 7.1.1 C Extrusion 23
- 7.2 2006 23
  - 7.2.1 E Marbles 23
- 7.3 2007 23
  - 7.3.1 B Mobiles Alabama 23
  - 7.3.2 C Out Of Sight 24
  - 7.3.3 D Witness Redaction 24

7.4 2008 . . . 24
7.4.1 G Stems Sell . . . 24
7.5 2010 . . . 24
7.5.1 C Cells . . . 24
7.6 2011 . . . 24
7.6.1 B Raggedy, Raggedy . . . 24
7.6.2 F Line of Sight . . . 24
7.6.3 H Road Rally . . . 25

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 to get access to the repository.

The following authors have contributed to this book in its current form:

- Godmar Back `<godmar@gmail.com>`

Chapter 1

Standard Libraries

The ACM ICPC, at the time of this writing, allows the use of the JDK 1.6 libraries as well as the C++ STL. Knowledge and mastery of the existing JDK classes is crucial for success. This chapter reviews some of the more commonly used classes.

1.1 Collection Classes

1.1.1 Object Equivalence

Java allows you to define and equivalence relationship between objects using the `Object.equals` method, which is required for some problems. If implemented, a conforming `hashCode` method must be implemented as well. Here are some hints.

- **Avoid gratuitous implementations.** In the vast majority of cases, you will not need your own `equals/hashCode` function! You only need equals/hashCode if your object is used as a key (not value!) in a Map or Set, **and** if the default implementation of equals ("two objects are equal if they are the same") does not suffice. Simply needing to store objects in a set, or using them as a map key, is not a justification for implementing equals(). An example of where equals is needed are search problems in a state space - you may create a state instance that is equal to an already explored state kept track of in a 'visited' set or map.

- **Implement equals() correctly**. You need to compare all relevant fields to one another.

```java
@Override
public boolean equals(Object _that) {
    State that = (State)_that;
    return this.field1 == that.field1 && ... && this.fieldn == that.fieldn;
}

// or, as appropriate
@Override
public boolean equals(Object _that) {
    State that = (State)_that;
    return this.field1.equals(that.field1) && ... && this.fieldn.equals(that.fieldn);
}
```

  For clarity, I recommend for `equals()` to always consider all fields, and to not include irrelevant fields in the object. (That is also why I like separate previous hop maps rather than previous hop fields, or distance fields in BFS implementations, see Section 3.2.1.) The problem with selectively including some fields in the comparison, and not others, is that it opens the door to mistakes in which you accidentally use information from the wrong object.

- **Understand the equals/hashCode contract**. The contract says two objects that are equal must have the same hashCode(). It does not require that if two objects have the same hashCode() they must be equals().
- **Implement hashCode() efficiently.** Rely on built-in functions, such as Arrays.hashCode() for arrays, or the built-in hashCode() functions for Strings and Lists, which are known to be good.
- **Consider hashCode()'s distribution.** The contract would be met by a degenerate function that returns always zero, but this would create terrible performance - in a hash map, all objects would be mapped to the same slot, resulting in linear lookup performance. The chosen hash function should provide a distribution of hash values that is as uniformly random as possible. Remember that Integer.hashCode does not use a Fowler-Noll-Vo hash, but rather returns the integer itself; the resulting randomness will be small, especially if the number of items added dwarfs the range in which that integer varies.

  When combining the hashes of fields, prefer bitwise xor (ˆ) over addition or multiplication. As an aside, it is not necessary that the lower bits are uniformly distributed - Java's HashMap implementation does not perform a simple modulo, but rather draws from higher-order bits, too:

```java
/**
 * Applies a supplemental hash function to a given hashCode, which
 * defends against poor quality hash functions.  This is critical
 * because HashMap uses power-of-two length hash tables, that
 * otherwise encounter collisions for hashCodes that do not differ
 * in lower bits. Note: Null keys always map to hash 0, thus index 0.
 */
static int hash(int h) {
    // This function ensures that hashCodes that differ only by
    // constant multiples at each bit position have a bounded
    // number of collisions (approximately 8 at default load factor).
    h ^= (h >>> 20) ^ (h >>> 12);
    return h ^ (h >>> 7) ^ (h >>> 4);
}
```

  In my 2011/H Solution 7.6.3, execution time reduced from 87 to 29 seconds by changing the hash function to use xor of all three components instead of multiplication of two components.

1.1.2 Comparators

The most common uses of comparators are for sorting (Arrays.sort, Collections.sort, Collections.max, etc.), binary trees (java.util.TreeMap), and sorted queues (java.util.PriorityQueue). Each use entails pitfalls.

- **Don't confuse partial orders with equality.** The contract of compareTo requires that the relationship be transitive across $<$, $>$, and $=$. That implies, for instance, that if $a > b \wedge b = c$ then $a > c$. When dealing with partial orders do not treat incomparable elements as equal. Notably, in a partial order, it's well possible that $a < c$ if $a > b \wedge \neg(b < c \vee b > c)$ Instead, either complete the order or use topological sorting.
- **Make the ordering consistent with equals.** A comparator should return 0 iff .equals() returns true. This is required, in particular, when using TreeSets. See compareTo.

Chapter 2

Simulation

While many simulation problems can be solved ad-hoc, some require, or may benefit from, a more principled approach.

2.1 Monte Carlo Methods

In some cases, random sampling might provide a solution. For instance, the code below computes π by examining whether randomly produced (x, y) samples are inside or outside the unit circle in the northeast quadrant.

```
import java.util.*;

public class Pi
{
    public static void main(String []av) {
        Random r = new Random();
        int incircle = 0, tries = 0;
        for (int i = 0; i < 20000000; i++) {
            double x = r.nextDouble();
            double y = r.nextDouble();
            if (x * x + y * y <= 1.0)
                incircle++;
            tries++;
        }
        System.out.printf("Pi is %f (exact value is %f)%n",
                4*incircle/(double)tries, Math.PI);
    }
}
```

Precision is generally not very good, mainly due to the statistical properties of the underlying pseudo-random number generator (PRNG). Increasing the number of samples will yield diminishing returns quickly. As a rule of thumb, don't expect more than 4 significant digits. For example, 2010/Cells 7.5.1 asks for a percentage with 2 digits after the period, for a total of 4 significant digits, making this approach feasible.

2.2 Discrete Event Simulation

Discrete Event Simulation is widely used to simulate physical phenomena, especially those that involve periodic and/or random events, as well as when the occurrence of future events depends on the state of the simulated world, rather than being fixed beforehand.

The key idea is to maintain a timeline as a queue of future events, sorted by their timestamp. Simulation time, aka virtual time, advances in jumps when the next upcoming event is pulled off the queue. Time jumps over period in which no events are scheduled (unlike a continuous simulation in which time increases in small, but constant, steps).

Event handlers may query the current time and they may schedule future events. Discrete Event simulations contain a typical core: a way to represent events, a queue to sort them, and an event loop that processes the event queue until either the maximum simulation time is reached or a solution to the problem is found. Recommended implementation strategy is to use immutable event objects that are inserted when scheduled and discarded after dispatch.

Example: UVA 00161 is an example of a problem which can be solved with DES. Note, however, that the small simulation time frame (18,000 seconds) also allows a continuous simulation approach, simply simulating every second, rather than only those points in time when a traffic light changes. This is a very simple example, with only 3 event types, all implemented in one class LightChange.

```
import java.util.*;
/**
 * Discrete Event Simulation.
 * UVA 00161 Traffic Lights
 *
 * @author Godmar Back
 */

public class Main
{
    /** Begin generic discrete event code. */
    Queue<Event> evQueue = new PriorityQueue<Event>();
    int currentTime;

    void schedule(Event e) {
        evQueue.offer(e);
    }

    int now() {
        return currentTime;
    }

    class Event implements Comparable<Event> {
        int time;
        Runnable what;

        Event(int time, Runnable what) {
            this.time = time;
            this.what = what;
        }

        @Override
        public int compareTo(Event that) {
            return this.time - that.time;
        }
    }

    /**
     * Event loop.
     *
     * Mostly generic, some problem-specific output interspersed.
     * Note that 'time' always jumps to the next event, rather than
     * being incremented one by one.
     */
    void simulate(int maxTime) {
        while (evQueue.peek().time <= maxTime) {
            Event e = evQueue.poll();
            currentTime = e.time;
            e.what.run();

```

```java
            if (!allGreen())
                continue;

            System.out.printf("%02d:%02d:%02d%n", e.time/3600, (e.time % 3600)/60, e.time % 60);
            return;
        }
        System.out.println("Signals fail to synchronise in 5 hours");
    }

    /** End generic discrete event code. */

    class LightChange implements Runnable {
        int light;
        LightState state;

        LightChange(int light, LightState state) {
            this.light = light;
            this.state = state;
        }

        @Override
        public void run() {
            states[light] = state;
            switch (state) {
            case GREEN:
                schedule(new Event(now() + cycles[light] - 5, new LightChange(light, LightState.ORANGE)));
                break;
            case ORANGE:
                schedule(new Event(now() + 5, new LightChange(light, LightState.RED)));
                break;
            case RED:
                schedule(new Event(now() + cycles[light], new LightChange(light, LightState.GREEN)));
                break;
            }
        }
    }

    enum LightState {
        GREEN, ORANGE, RED;
    };

    LightState [] states;
    int [] cycles;

    Main(List<Integer> lightcycles) {
        states = new LightState[lightcycles.size()];
        cycles = new int[lightcycles.size()];
        for (int i = 0; i < lightcycles.size(); i++) {
            states[i] = LightState.GREEN;
            cycles[i] = lightcycles.get(i);
            /* prime event queue with when the traffic lights first turn orange */
            schedule(new Event(cycles[i] - 5, new LightChange(i, LightState.ORANGE)));
        }
    }

    boolean allGreen() {
        for (LightState s : states) {
            if (s != LightState.GREEN) {
                return false;
            }
        }
        return true;
    }

    public static void main(String []av) {
        Scanner s = new Scanner(System.in);
        for (;;) {
            ArrayList<Integer> lightcycles = new ArrayList<Integer>();
```

```
            int n;
            while ((n = s.nextInt()) != 0)
                lightcycles.add(n);

            if (lightcycles.size() == 0)
                return;

            new Main(lightcycles).simulate(5 * 60 * 60);
        }
    }
}
```

Chapter 3

Searching

3.1 Backtracking

3.2 State Space Exploration

In these problems, an initial state is to be transformed through a series of valid moves into one goal state, or possibly one of multiple possible goal states. Examples include block puzzles or single-player games. These problems have the following characteristics:

- An optimal solution required: we want the minimal number of moves from the initial state to goal state, rather than just any.
- No obvious strategy. Given a state, there's no obvious way to choose which move to make to get closer to the goal. In fact, it's usually even difficult to tell how far we might be away from the goal state. Any easy-to-find lower bounds for this distance to the goal might be far too optimistic. See also A^* in Section 3.2.2.

3.2.1 Breadth-First Search (BFS)

There are numerous ways of writing a BFS loop. There are even more ways of getting it wrong. Here is one possible way:

```
/** BFS Skeleton.
 * Assumes that 'State' implements equals() and hashCode()
 * according to contract.
 * State must also provide 'isfinal', and 'successors' methods
 */
void solve(State start) {
    Set<State> visited = new HashSet<State>();
    // has this state been visited?
    Map<State, State> pred = new HashMap<State, State>();
    // predecessor on the shortest path to the start state
    Map<State, Integer> dist = new HashMap<State, Integer>();
    // shortest distance to start state
    Deque<State> bfs = new ArrayDeque<State>();  // BFS queue
    bfs.offer(start);
    dist.put(start, 0);

    while (bfs.size() > 0) {
        State s = bfs.poll();
        int n = dist.get(s);
```

```
        visited.add(s);

        if (s.isfinal()) {
            output(n, s, pred);
            return;
        }

        for (State succ : s.successors()) {
            if (visited.contains(succ))
                continue;

            if (!pred.containsKey(succ))
                pred.put(succ, s);

            if (!dist.containsKey(succ)) {
                dist.put(succ, n+1);
                bfs.offer(succ);
            }
        }
    }
}

/* Compute and output path */
void output(int distToSolution, State finalState, Map<State, State> pred) {
    System.out.println("The distance to the solution is: " + distToSolution);

    List<State> revPath = new ArrayList<State>();
    State s = finalState;
    while (pred.containsKey(s)) {
        revPath.add(s);
        s = pred.get(s);
    }
    revPath.add(s);

    for (int i = 0; i < revPath.size(); i++) {
        System.out.printf("%3d %s%n", i, revPath.get(revPath.size() - 1 - i));
    }
}
```

Notes.

- Adding the final state could be avoided by calling output() if a final state is found, potentially saving the unnecessary expansion of states if the optimal goal state is already in the queue - however, then the case where the initial state is final must be handled separately. Seen in 2006/E Marbles 7.2.1 where the judge data contains the case that the initial state is final.

- The state class should use ducktyping and provide the necessary methods - isfinal, successors. In addition, the state class must implement an object equivalence relationship (equals, hashCode) as described in Section 1.1.1.

- The example keeps track of both the path (via 'pred') and distance (via 'dist'). If the problem asks for only one of the two, the other can be omitted. In that case, the insertion of a successor state should be guarded by the one remaining. Note that both 'pred' and 'dist' have the invariant when encountering a state multiple times, only the first encountered state is kept in the map.

When Not To Use BFS. There are some problems that may appear to be solvable using BFS, but are in fact dynamic programming problems. These are generally problems in which there are many transitions from states farther in the problem space to states that have been discovered much earlier. An example is 2007/C/Out of Sight. 7.3.2.

3.2.2 A^*

A^* [2] is a classic algorithm to improve upon breadth-first search when an estimator function for states that estimates their distance to the goal state is known. Though an AI algorithm, it can yield guaranteed optimal solution if the estimator is chosen carefully.

The idea is simple. In BFS, we're examining nodes based on their distance from the start state. So we'll look at, and expand, all states that are 6 moves away from the start *before* we look at any state that's 7 moves from the start - simply because of the FIFO discipline in the queue.

In A^*, instead of putting all to-be-explored states in a FIFO queue, we sort them by their "goodness," which is defined as a lower bound of how many moves a solution is away from the state. For instance, if a state A is 6 moves away from the start, and it is known that it is at least 10 moves away from the goal, it would have a goodness of 16 (6 + 10). On the other hand, if a move B is 8 moves away from the start, and at least 4 moves away from the solution, its goodness is 12 and it's explored first. Note that we are not estimating how far the state is from the solution. We're simply constructing a function that says: this state is *at least* this far from a solution. It may be farther - our decision to explore state B may be wrong and the closest solution state may result from the exploration of A.

The challenge lies in how to say - quickly - that a state is "at least this many moves away from the goal." In general, that's tough since if we knew how many states a state is away from the goal, it would be much easier to solve the problem. However, there are some lower bounds that can be found - for instance, in a puzzle, we could count how many pieces are not in their final position, knowing that it'll take at least this many moves to get them there (assuming it's a puzzle like the traditional 15-piece puzzle where all pieces need to be sorted). Another possible function is the sum of the Manhattan distances between each piece's current and final position. Again, it is clear that it will take at least this many moves.

In competition problems you are practically never asked to find an 'almost' perfect heuristic solution - they ask you to compute the correct, and in search problems like these, the shortest solution. A^* will provide you with the optimal solution if the estimation function does not overestimate - if it did, it might lead to the nearest goal state being placed behind a farther goal state in the queue. That farther state might then be discovered first, and mistaken for the shortest solution.

It's not a problem that the estimation function underestimates - in fact, it'll generally do that - this just means that A^*might waste some time exploring states it thinks are promising, but which do not actually lead to a solution.

Converting a BFS to A-Star in Java is really simple. Replace the entry of the BFS described in Section 3.2.1 such that the BFS work queue uses a PriorityQueue like so:

```java
// shortest distance to start state
final Map<State, Integer> dist = new HashMap<State, Integer>();
Queue<State> queue;
if (useAstar) {
    // sort by sum of distance to start state + mindist to goal
    queue = new PriorityQueue<State>(1000, new Comparator<State>() {
        @Override
        public int compare(State a, State b) {
            int ascore = dist.get(a) + a.mindistancetogoal();
            int bscore = dist.get(b) + b.mindistancetogoal();
            return Integer.valueOf(ascore).compareTo(bscore);
        }
    });
} else {
    queue = new ArrayDeque<State>();                  // BFS queue
}
queue.offer(start);
```

Note how making dist final allows it to be used in the comparator. The code exploits that both PriorityQueue and ArrayDeque implement the Queue interface.

Notes

- A^*will increase the cost of inserting and removing a state from the queue substantially - in a heap-based priority queue, 'offer()' and 'poll()' take O(log n) whereas they are constant time O(1) operations on a Deque. On the other hand, especially if the estimator function is too optimistic, the state space may only be marginally reduced. Recommendation: use A-Star only if BFS times out and/or if a good heuristic can be found.

Chapter 4

String Processing

4.1 Regular Expressions

4.1.1 Regular Expressions

4.1.2 Zero-width Lookahead Technique

In some problems (notably, 2007/B/Mobile 7.3.1, 2007/D/Witness 7.3.3, and 2008/G/Stems 7.4.1), the string and/or input handling of these problems can greatly benefit from using zero-width positive lookahead/lookbehind regular expressions.

To understand how they work, consider how java.util.Scanner works. By default, a Scanner splits the input stream into tokens using a delimiter pattern. The default delimiter pattern is one or more whitespaces (written as `\p{javaWhitespace}` or, when embedded in Java code, as `"\\p{javaWhitespace}+"`). The input characters that are matched by the delimiter itself are consumed by the Scanner there is no way to retrieve them.

In some cases, whitespace is not a suitable delimiter. Suppose you're asked to parse an arithmetic expression that uses +, -, *, and /. Whitespaces are optional, so both `1+1` and `1 + 1` as well as `1 +1` are valid expressions. If you made the operators '+', '-' etc. delimiters (perhaps in addition to whitespace), a Scanner would retrieve '1' and '1', but there would be no way to retrieve the '+' so you couldnt distinguish '1+1' and '1-1'. Instead, use lookaround matching by adding a zero-width delimiter that matches before or after a +, -, *, or /. "Zero-width" here means that although the delimiter matches (and thus causes the Scanner to stop and return what it has read so far!), it does not consume any characters. Thus, the scanner will stop,

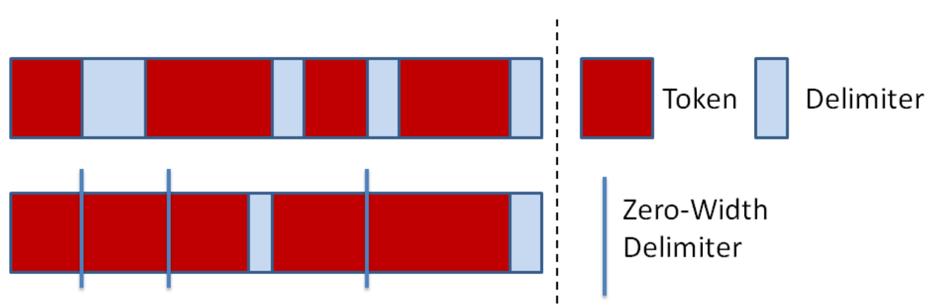

Figure 4.1: Lookahead Splitting. The top shows a traditional scanner/split which consumes delimiters. The bottom shows a scanner using delimiter expressions that may or may not consume characters.

but the delimiter (which the Scanner swallows) has zero width therefore, the characters are returned as part of the previous token. In this example, s.next() would return '+'.

Figure 4.1 shows a traditional scanner (top) and a scanner that uses both consuming and non-consuming delimiters (bottom): if the delimiter used by the scanner does not consume any characters, the scanner will return the entire input stream. This is very useful if you need to manipulate a stream without losing any characters.

The idea to use String.format to turn any regular expression into a zero-width lookahead or lookbehind delimiter is taken from here.

Note that this technique can be used with a java.util.Scanner object (via useDelimiter), but also in all other functions that use regular expressions as delimiters, notably String.split().

Finally, note that you cannot use some regular expressions to describe zero-width delimiters. Notably, **expressions using repetition (* or +) cannot be used.**

Code Example. The following program shows some of the applications of this style of matching. These examples include:

- Arithmetic expressions with optional whitespace
- S-Expressions with optional whitespace before and after ( )
- Finding words in a sentence
- Finding sentences in a paragraph

```java
/**
 * Examples of zero-width lookahead/lookbehind splitting.
 * For each example, study the input and output.
 *
 * http://stackoverflow.com/questions/2206378/how-to-split-a-string-but-also-keep-the-delimiters
 *
 * @author Godmar <godmar@gmail.com>
 */

import java.util.*;

public class Lookaround
{
    /* String.format patterns for ease of use */
    final static String MATCH_BEFORE_OR_AFTER = "((?<=%1$s)|(?=%1$s))";
    final static String MATCH_AFTER = "(?<=%1$s)";
    final static String MATCH_BEFORE = "(?=%1$s)";

    static void example(String input, String delim) {
        Scanner s = new Scanner(input).useDelimiter(delim);
        System.out.println("Delimiter: " + delim);
        System.out.println("Input: " + input);
        System.out.print("Output: '" + s.next() + "'");
        while (s.hasNext()) {
            System.out.print(", '" + s.next() + "'");
        }
        System.out.println();
        System.out.println();
    }

    public static void main(String []av) {
        // match right before or after +, -, *, /
        // consumes nothing
        String delim = String.format(MATCH_BEFORE_OR_AFTER, "[\\+\\-\\*\\/]");
        example("10+21*32-43/5+60", delim);

        // matches whitespace or  right before or after +, -, *, /
        // consumes whitespace - but no +/-/*//
```

```
        delim = "\\p{javaWhitespace}|"
            + String.format(MATCH_BEFORE_OR_AFTER, "[\\+\\-\\*\\/]");
        example("10 + 21*32 -43 / 5+60", delim);

        // match whitespace or right before or after ( )
        // consumes whitespace - but does not consume ( or )
        delim = "\\p{javaWhitespace}|"
            + String.format(MATCH_BEFORE_OR_AFTER, "[\\(\\)]");
        example("((F1 (A1 1 2 3)) (F2 (A2 4 5) ( A3 5 )))", delim);

        // match at word boundaries
        // consumes nothing
        delim = "\\b";
        example("This text has words, and some---wrongly set---punctuation characters."
            +"Note that words can contain alphanumericals such as babe1234 and"
            +" underscores.  Underscores do_not_form_a_word_boundary.", delim);

        // match at before or after anything that is not alphanumeric
        // (which matches every boundary except within a word of alphanumeric chars.)
        // consumes nothing
        delim = String.format(MATCH_BEFORE_OR_AFTER, "[^A-Za-z0-9]");
        example("This delimiter identifies word boundaries, but unlike the previous "
            +"one returns all characters between words as individual tokens. "
            +"Underscores do_form_a_word_boundary with this delimiter.",
             delim);

        // match after ., !, ? or empty line.
        delim = String.format(MATCH_AFTER, "[\\.\\!\\?]|\\n\\n");
        example("This matches sentences. And questions too?  Yes!  "
            +"Even breaks between\n\nparagraphs.", delim);
    }
}
```

4.1.3 NFA Simulation

The Regex engine in Java does not convert to a Thompson-DFA [1]; 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. Figure 4.2 shows an example of how to construct such a NFA. In an NFA there may be multiple transitions labeled with the same symbol: for instance, there's a transition labeled 'b' from state 0 to state 1, but there is also a transition labeled 'b' from state 0 to state 0. For the input string `abc`, the 'b' would transition into state 1, whereas for the input string `abbc`, the first 'b' would transition into state 0, the second into state 1.

Of course, we don't know which it's actually going to be - a NFA, in its theoretical formulation, is defined to pick the correct transition, like an Oracle would. That's why we simulate it by simply keeping track of all possible ("active") states the NFA might be in after each symbol. This is done using a set (HashSet or BitSet if the states are nicely numbered). For each input symbol, we compute the possible set of successor states based on the current set of active states. If after the string has been exhausted the goal state is in the set of active states the string is matched. A Python solution is shown below for succinctness.

```python
import sys, string
from collections import defaultdict

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

```

[1] See [1] for background on Thompson's idea

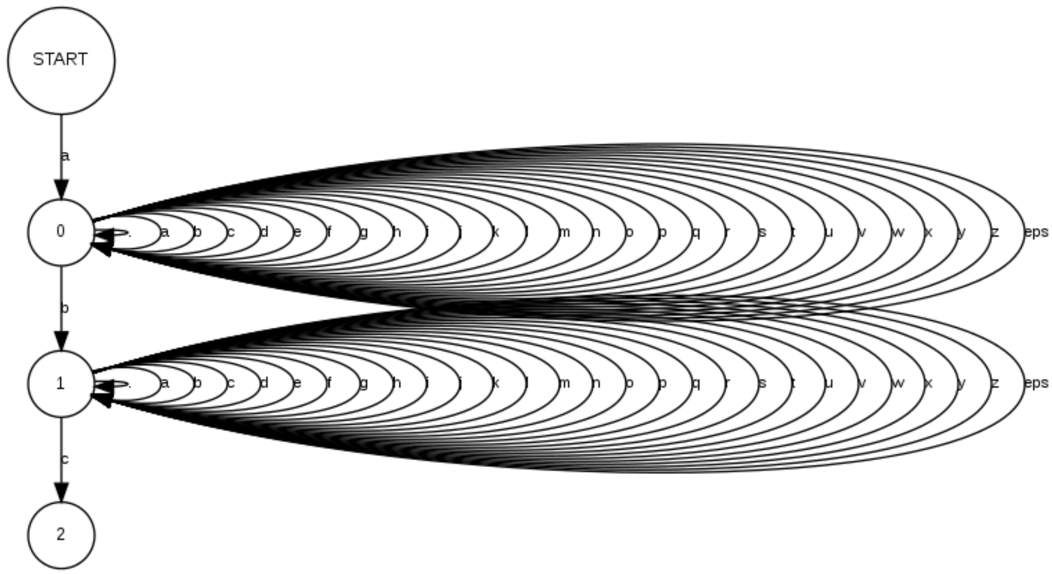

Figure 4.2: NFA for regular expression `a.*b.*c` representing glob `a*b*c` over alphabet of lowercase letters and period (.)

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

4.2 Parsing

In programming problems, reading the input generally does not require parsing in the sense that the syntactic structure of the input is given as a context-free grammar. When it does, it is usually part of the problem's challenge. Though parsing, in general, is a wide topic area - the cases occurring in programming contests are usually simple grammars that describe a LL(1) language, and the grammar is typically included in the problem's specification.

4.2.1 Recursive Descent

In a recursive descent parser, the structure of the code mirrors the structure of the grammar. Non-terminals are represented by functions. For instance, to read an equation which represents two expressions separated by a '=' terminal, the grammar rule

$$Equation \to Expression = Expression$$

would be implemented by a method such as this:

```
ASTNode parseEquation() {
    ASTNode lhs = parseExpression();
    if (lookahead() != '=')
        throw new Error("Missing =");
    tokens.poll();  // consume token
    ASTNode rhs = parseExpression();
    return new Equation(lhs, rhs);
}
```

`lookahead()` must peek at the next input token, but must not consume it. A parser may call lookahead() multiple times before committing to nonterminal and consuming the token. Generally, a recursive-descent parser's methods return a data structure that represents the nonterminal implemented by that function. Subtyping is commonly used to handle alternatives.

Chapter 5

Geometry

5.1 Basics

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

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

5.2 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.

5.3 Coordinate Geometry

5.3.1 Line/Line Intersection

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

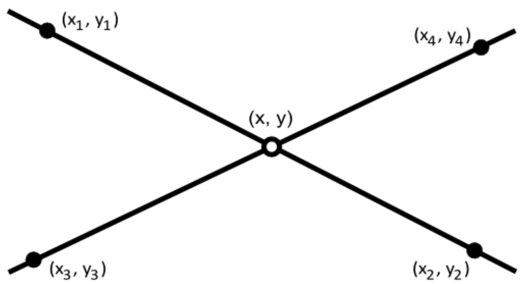

Figure 5.1: Line line intersection

The determinants can be written out as:

$$(P_x, P_y) = \Bigg(\frac{(x_1 y_2 - y_1 x_2)(x_3 - x_4) - (x_1 - x_2)(x_3 y_4 - y_3 x_4)}{(x_1 - x_2)(y_3 - y_4) - (y_1 - y_2)(x_3 - x_4)}, \frac{(x_1 y_2 - y_1 x_2)(y_3 - y_4) - (y_1 - y_2)(x_3 y_4 - y_3 x_4)}{(x_1 - x_2)(y_3 - y_4) - (y_1 - y_2)(x_3 - x_4)} \Bigg)$$

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

Notes

- Does not handle parallel or coincident lines: Denominator will be zero:

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

- Does not handle if lines are each others' normal (i.e., at a right angle). If line is horizontal ($y_1 = y_2$ or $y_3 = y_4$), and the other vertical ($x_1 = x_2$ or $x_3 = x_4$) denominator will also be 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 This code is from a solution to 2011/F (Section 7.6.2) where the parallel and rectangular cases do not occur. (TBD: provide complete implementation.)

```
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 and are not at right angle)
    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);
}
```

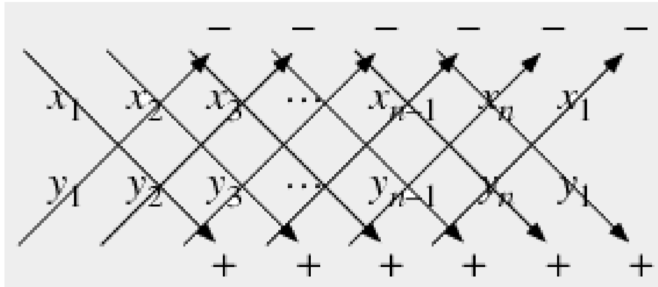

Figure 5.2: Line line intersection

5.3.2 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 5.2 shows how to multiply this out

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

(Source: Mathworld [3])

Notes

- Works for any simple polygon (concave or convex)
- Does not work for complex polygons (when any edges intersect)
- Points **must be ordered** if polygon has more than 3 vertices, or output is junk.
- A is positive if points are in counterclockwise order, negative if points are in clockwise order. See the use of Math.abs() in code below.
- Triangle and any Quadrilateral are, of course, just special cases. For triangles, order does not matter.

Code

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

Chapter 6

Gotchas

Common mistakes and idiosyncrasies observed in the judge input and specification of various problems posed at competitions.

1. **Judge input not terminated as required**. Typically, the problem states that there's some way to identify the end of input without having to rely on EOF. We've observed judge input, however, where EOF terminated the input. You should try to write your input loop such that your solution works whether the input is terminated by EOF or by the specified end-of-input delimiter. This strategy may allow you to submit a correct solution even before the mistake is discovered (and may even lead to a delay in when it's discovered that would benefit your team.) Seen in 2006/E Marbles 7.2.1.

2. **Insignificant Trailing Spaces**. In problems that state ”there is one word per line” we have observed trailing spaces which must be trimmed. Advice: always use String.trim(), unless the spaces are significant, which we have not seen anywhere. Seen in 2007/D Witness 7.3.3.

3. **Insignificant Leading Spaces**. In some problems, (insignificant) leading spaces may occur. The catch here is that naive splitting without trimming may produce an empty string in the first position. See bsh output below:

   ```
   % System.out.println(Arrays.toString("  word1  word2  ".split("\\s+")));
   [, word1, word2]
   ```

   Seen in 2011/B Raggedy 7.6.1.

4. **Significant Leading and Trailing Spaces**. Check if the problem allows for significant leading and trailing spaces, such as when reading in grids or mazes or specified width/height. Don't accidentally trim(), but don't expect the trailing spaces, either. Consider allocating and filling an array of desired length and using System.arraycopy to copy the input.

   Seen in 2011/H Road Rally 7.6.3.

Chapter 7

Mid-Atlantic Problem Sets

This chapter contains some notes about the problems occurring in the Mid-Atlantic problem set. We focus on this corpus in particular because there are recurring themes since the problems have been created by the same person (or team) for multiple years.

7.1 2005

(Problem Set PDF 2005)

7.1.1 C Extrusion

Straightforward application of polygon area formula, see Section 5.3.2.

7.2 2006

(Problem Set PDF 2006)

7.2.1 E Marbles

A simple state space exploration problem solvable with straightforward BFS exploration. Catch: judge input data missed the "0 0 0" line.

7.3 2007

(Problem Set PDF 2007)

7.3.1 B Mobiles Alabama

Lexical analysis benefits from zero-width lookaround 4.1.2, although simpler solutions may work, too, such as replacing '(' and ')' with ' (' and ') ' before splitting on whitespace. Recursive descent parsing should be used to analyze the syntactical structure of the input.

7.3.2 C Out Of Sight

Dynamic programming.

Applying BFS will time out due to an explosion in the number of successor states, most of which will have been already seen.

7.3.3 D Witness Redaction

This problem can be solved with regular expressions and zero-width lookaround splitting. See Section 4.1.2.

7.4 2008

(Problem Set PDF 2008)

7.4.1 G Stems Sell

Can be solved with regular expressions.

Judge data appears broken, even on ICPC site.

7.5 2010

(Problem Set PDF 2010)

7.5.1 C Cells

Solvable using Monte Carlo simulation. Can use java.awt.geom.* to implement containment check. Also solvable using numerical integration (in polar coordinates).

7.6 2011

(Problem Set PDF 2011)

7.6.1 B Raggedy, Raggedy

This problem can be solved using dynamic programming. Note that there may be leading spaces on some input lines.

7.6.2 F Line of Sight

Straightforward application of area of polygon 5.3.2 and line intersection 5.3.1. Note that parameters of the problem even exclude corner cases for line intersection (e.g. parallel lines, right angles).

7.6.3 H Road Rally

Straightforward state space exploration using BFS, see Section 3.2.1.

Input handling requires care since handout is misleading by showing race tracks enclosed in 'x' which is not required. Also, the 'horizontal' and 'vertical' distance is to be treated signed, not absolute.

Bibliography

[1] Russ Cox. Regular expression matching can be simple and fast (but is slow in Java, Perl, PhP, Python, Ruby, ...). `http://swtch.com/~rsc/regexp/regexp1.html`, 2007.

[2] Judea Pearl. *Heuristics: intelligent search strategies for computer problem solving*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1984.

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

Index

Area
    Polygon, 19

Line/Line Intersections, 17

NFA
    Simulation, 13

Polygon, 19
    Area, 19