Robert Koeninger

CS471 Design and Analysis of Algorithms
Take Home Final Exam
Wednesday, August 27, 2008

**Problem 1:** Determine the average complexity of quicksort for an array of length n = 3.

The number of comparisons was counted by executing a program implementing the algorithm as shown in the textbook and outputting the number of comparisons to a file.

| Input Array | Comparisons Made |
|---|---|
| {1, 2, 3} | 5 |
| {1, 3, 2} | 5 |
| {2, 1, 3} | 3 |
| {2, 3, 1} | 3 |
| {3, 1, 2} | 5 |
| {3, 2, 1} | 5 |
| Average: | 4.333 |

The algorithm as implemented to determine results:

```
int comparisons; // A global variable to keep track of the number of comparisons

/*
 * The quicksort’s partition function, as shown in the book
 */
int partition(int* array, int low, int high){
       int left = low, right = high;
       int pivotValue = array[low];
       int temp;
       while (left < right){
              while ((array[left] <= pivotValue) && (left <= high)){ // Comparison made here…
                     left++;
                     comparisons++; // Increment comparison counter
              }
              while (array[right] > pivotValue){ // And comparisons made here
                     right--;
                     comparisons++; // Increment comparison counter
              }
              if (left < right){
                     temp = array[right];
                     array[right] = array[left];
                     array[left] = temp;
              }
       }
       array[low] = array[right];
       array[right] = pivotValue;
       return right;
}

/*
 * The recursive quicksort function
 */
void quicksort(int* array, int low, int high){
       if (high > low){
              int position = partition(array, low, high);
              quicksort(array, low, position - 1);
              quicksort(array, position + 1, high);
       }
}
```

**Problem 2:** Design a divide and conquer algorithm for finding the maximum and minimum elements in an array.

The algorithm works by recursively splitting the array in half until it has branched down to a sub-list of only one element. In an array of one element, that element is the minimum and maximum in the list. As the recursive calls return, the function compares the minimum and maximum of each branch until the function returns to its root, where the maximum of each half is compared to find the maximum of the entire array and the minimum of each half is compared to find the minimum of the entire array.

In pseudocode:

```
maxmin(array, low, high)
        if (high = low)
                retmax = array[low]
                retmin = array[low]
        middle = (low + high) / 2;
        max1, min1 = maxmin(array, low, middle);
        max2, min2 = maxmin(array, middle + 1, high);
        if (max1 > max2)
                retmax = max1
        else
                retmax = max2
        if (min1 < min2)
                retmin = min1
        else
                retmin = min2
```

An implementation in C:

```c
/*
 * The maxmin function. Returns an array with 2 elements. The first is the maximum of the
 * specified region of the array, and the minimum is the second element.
 */
int* maxmin(int* array, int low, int high){
        int* res = new int[2];
        if (high == low){
                res[0] = res[1] = array[low];
                return res;
        }
        int middle = high > low + 1 ? (low + high) / 2 : low;
        int* res1 = maxmin(array, low, middle);
        int* res2 = maxmin(array, middle + 1, high);
        res[0] = res1[0] > res2[0] ? res1[0] : res2[0];
        res[1] = res1[1] < res2[1] ? res1[1] : res2[1];
        delete[] res1;
        delete[] res2;
        return res;
}
```

**Problem 3:** Pan's matrix multiplication method requires that n = $70^k$ and results in the recurrence relation T(n) = 143,640 * T(n/70). With T(1) = 1, show that T(n) = O($n^{2.795}$).

$$T(n) = 143640 * T(n/70)$$
$$T(70) = 143640 * T(1)$$
$$T(70^2) = 143640^2 * T(1)$$
$$T(70^3) = 143640^3 * T(1)$$
$$T(70^k) = 143640^k * T(1)$$
$$T(70^k) = 143640^k$$
$$k = \log_{70} \mathrm{n}$$
$$T(n) = 143640^{\log_{70} n}$$
$$\log_{143640} T(n) = \log_{70} n$$
$$\frac{\log T(n)}{\log 143640} = \frac{\log n}{\log 70}$$
$$\log T(n) = \log n * \frac{\log 143640}{\log 70}$$
$$\log T(n) = \log n * 2.795$$
$$\log T(n) = \log n^{2.795}$$
$$T(n) = O(n^{2.795})$$

**Problem 4:** Generate 3-CNF's at random and plot the probability that the CNF will be satisfiable as m changes for a given n. Do this at a minimum of 3 values for n.

My program could not generate a 3-CNF that would fail, or always found a 3-CNF to be true. Code attached.

Results:

| | | | |
|---|---|---|---|
| n = | | 1 | |
| m = | | 3 | 13 |
| m's: | | averages: | |
| | 3 | 1 | |
| | 4 | 1 | |
| | 5 | 1 | |
| | 6 | 1 | |
| | 7 | 1 | |
| | 8 | 1 | |
| | 9 | 1 | |
| | 10 | 1 | |
| | 11 | 1 | |
| | 12 | 1 | |
| | 13 | 1 | |
| n = | | 3 | |
| m = | | 3 | 13 |
| m's: | | averages: | |
| | 3 | 1 | |
| | 4 | 1 | |
| | 5 | 1 | |
| | 6 | 1 | |
| | 7 | 1 | |
| | 8 | 1 | |
| | 9 | 1 | |
| | 10 | 1 | |
| | 11 | 1 | |
| | 12 | 1 | |
| | 13 | 1 | |
| n = | | 5 | |
| m = | | 3 | 13 |
| m's: | | averages: | |
| | 3 | 1 | |
| | 4 | 1 | |
| | 5 | 1 | |
| | 6 | 1 | |
| | 7 | 1 | |

|  |  |
|---|---|
| 8 | 1 |
| 9 | 1 |
| 10 | 1 |
| 11 | 1 |
| 12 | 1 |
| 13 | 1 |

## Code:

```
import java.util.Arrays;

import java.util.Random;

import java.io.*;


public class CNF{

        public static void main(String[] args) throws IOException{

                PrintWriter outfile = new PrintWriter(

                new FileOutputStream(new File("cnf-results.csv")));

                int[] ns = {1,3,5};

                int ns_size = 3;

                int n;

                int m_min = 3, m_max = 13;

                int probTotal = 100;

                int successful;

                double avg;

                for (int a = 0; a < ns_size; ++a){

                        n = ns[a];

System.out.println("n=" + n);

                        outfile.println("n =," + n);

                        outfile.println("m =," + m_min + "," + m_max);

                        outfile.println("m's:,averages:");

                        for (int m = m_min; m <= m_max; ++m){

System.out.print("m="+m+" ");

                        successful = 0;

                                for (int k = 0; k < probTotal; ++k){
                                        if (getResultBruteforce(generateRandom3CNF(n,m))){
                                                successful++;
                                        }else{
System.err.println("---NOT SATISFIED---");
                                        }
                                }
                                avg = ((double)successful) / ((double)probTotal);
System.out.println("succ=" + successful + " avg=" + avg);
                                outfile.println(m + "," + avg);
                        }

System.out.println();
                }
                outfile.flush();
                outfile.close();
        }

        // Returns an array of boolean values that are a solution to the CNF
        // retval[0] is the value of X1, retval[1] is the value of X2, etc
        // returns null if no result found
        // currently uses a brute-force function
        static boolean getResultBruteforce(CNF cnf){
                cnf.valTable = new boolean[cnf.variableCount];
                for (long valueFlag = 0; valueFlag < (1 << cnf.variableCount);
                ++valueFlag){
//System.out.println(valueFlag);
                        for (int x = 0; x < cnf.valTable.length; ++x){
                                cnf.valTable[x] = (valueFlag & (1 << x)) != 0;
                        }
                        if (cnf.getValue()){
//System.out.println(cnf.toString());
//System.out.println(Arrays.toString(values));
```

```
                        return true;
                }
        }
        return false;
}

// Random 3-CNF generator for N number of clauses with
// M unique variables
// n = numClauses, m = numVars
static CNF generateRandom3CNF(int numClauses, int numVars){
        CNF cnf = new CNF();
        Random rand = new Random();
        cnf.clauses = new Clause[numClauses];
        cnf.variableCount = numVars;
        boolean[] varsUsed = new boolean[numVars];
        for (int x = 0; x < cnf.clauses.length; ++x){
                Arrays.fill(varsUsed, false);
                cnf.clauses[x] = new Clause();
                cnf.clauses[x].vars = new Variable[3];
                for (int y = 0; y < cnf.clauses[x].vars.length; ++y){
                        boolean looping = true;
                        cnf.clauses[x].vars[y] = new Variable();
                        while (looping){
                                int newVar = rand.nextInt(numVars) + 1;
                                if (! varsUsed[newVar - 1]){
                                        cnf.clauses[x].vars[y].id = newVar;
                                        cnf.clauses[x].vars[y].negated = rand.nextBoolean();
                                        varsUsed[newVar - 1] = true;
                                        looping = false;
                                }
                        }
                }
        }
        return cnf;
}

/* Class structure for generic CNF */

// list of clauses in the CNF
private Clause[] clauses;
private int variableCount;
private static boolean[] valTable;

boolean getValue(){
        for (int x = 0; x < clauses.length; ++x){
                if (!clauses[x].getValue())
                        return false;
        }
        return true;
}

public String toString(){
        String retval = "";
        for (int x = 0; x < clauses.length; ++x){
                for (int y = 0; y < clauses[x].vars.length; ++y){
                        retval += (clauses[x].vars[y].negated ? "-" : "");
                        retval += clauses[x].vars[y].id + " ";
                }
                retval += "\n";
        }
        return retval;
}

// each clause contains an array of variables
```

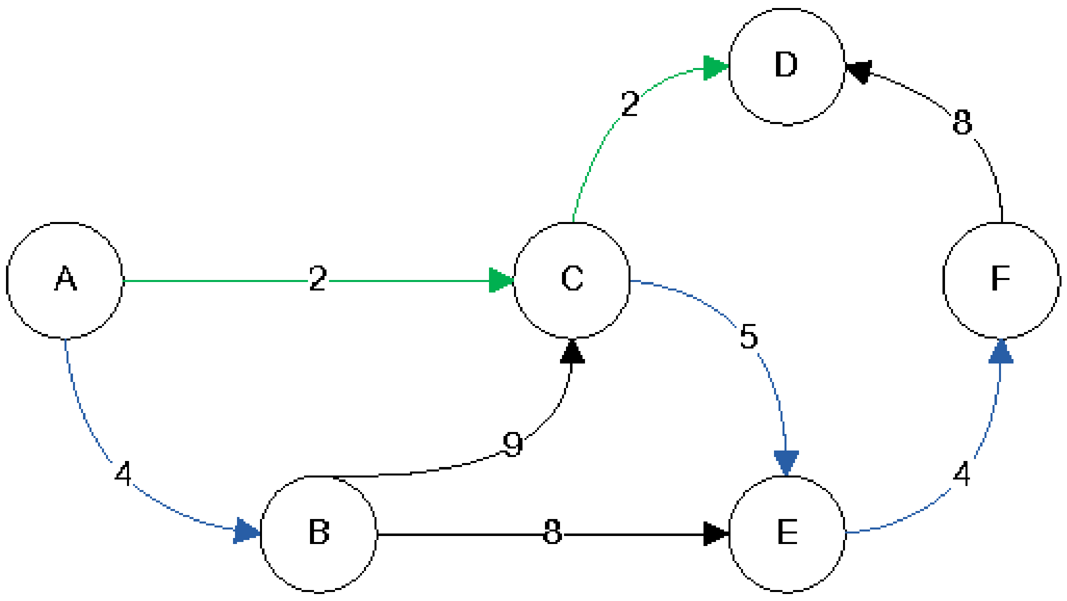
```
private static class Clause{

        Variable[] vars;

        boolean getValue(){

                for (int x = 0; x < vars.length; ++x){

                        boolean v = vars[x].getValue();

                        if (vars[x].negated){

                                v = !v;

                        }

                        if (v)

                                return true;

                }

                return false;

        }

}


// each variable has an id associated with it and if it is negated

// in the clause

private static class Variable{

        int id;

        boolean negated;

        boolean getValue(){

                return valTable[id - 1];

        }

}}
```

**Problem 5:** Show that Floyd’s algorithm works even when some of the edges have negative weights, provided that there is no negative weight cycle. What goes wrong when there is a negative weight cycle?

Floyd’s algorithm works by simple addition so edge weights can be entirely relative. Unlike Dijkstra’s algorithm, Floyd’s doesn’t move node to node looking for a cheaper path, it looks at the entire cost table every iteration, so negative values will be accounted for. If there is a negative weight cycle, the least-cost path between two nodes becomes infinitely small, and the algorithm has a constant, finite run time, so the distance cannot be properly measured. Negative weight cycles are not properly handled by the algorithm, nor can they be, as they have an infinitely small least-cost path.

**Problem 6:** Show by example that Dijkstra's algorithm does not generate a minimum spanning tree, that is, the shortest path tree does not have to be the same as the minimum spanning tree.

It is not the intended purpose of Dijkstra's algorithm to find the minimum spanning tree to every node on the graph. The algorithm greedily chooses the cheapest path until it reaches the destination node. In the figure below, the green arrows are the edges that Dijkstra's algorithm chooses (they are the cheapest), and the blue arrows are the parts of the minimum spanning tree that the algorithm misses.



**Problem 7:** Will the Bellman-Ford algorithm work when there is a link with a negative weight? What happens when there is a negative weight cycle?

The Bellman-Ford algorithm works with negative edge weights because it compares edge weights as relative values, not absolute ones. Unlike Dijkstra's algorithm, Bellman-Ford doesn't search the graph node by node, it looks at the entire table of edge weights at the same time. If there is a negative-weight cycle, the algorithm will not terminate correctly, because it will never be able to find a minimum cost, each time it iterates along a path of edges with a negative weight cycle, it will repeatedly find a cheaper and cheaper path. Many implementations of the Bellman-Ford algorithm include a check for negative cycles so that the input graph can be rejected outright and avoid this problem.

**Problem 8:** Show by example that Dijkstra's algorithm can fail when there is an edge with a negative weight.

In the graph shown below, Dijkstra's algorithm would start at node A. It would chose the path to node C (at a cost of 2) because it is cheaper than the path to node B. Then it would move from C straight to the destination node D. The least-cost path is actually A -> B -> C -> D, but Dijkstra's algorithm is designed to be greedy and select the path that appears the cheapest from the node it is currently visiting. Having negative values in unvisited regions of the graph allows for cheaper paths to appear along longer (more edges) paths of the graph. The red edges are the path that Dijkstra's algorithm takes, the blue path is the actual least-cost one.