

CS776
Assignment 0

Mark Harmer

September 9, 2009

Algorithm

Initially, I set out to randomly flip bits one at a time and determine if the result was lower than previously. If it was lower, it was discarded since it went "downhill", previous bits were reused and a new bit position was chosen. However, if the result was the same or higher the new bit vector was kept and the old discarded. My initial results using this method didn't solve the problem at all, the algorithm ran for a minute without solving the eval1 problem, so another approach was taken.

Looking at the bit vector the algorithm was generating it appeared to become stuck in a local optima; it was continually retrieving a value of 80.0 from the eval1 problem, yet only a few of the bits in the vector were being changed without going downhill. What I assumed was happening was it was still trying to solve the problem for all the bits in the vector when it should only be trying to solve those that it knows it won't be going downhill on. I implemented what I call "bit progression", which holds a boolean flag for each bit. If the bit is flipped and the result goes downhill then the flag is toggled and that bit can no longer be changed by the algorithm. This reduces the problem size quickly to 20 bits in the eval1 problem. From these 20 bits the algorithm randomly flips one each time and eventually finds the solution in a few seconds.

Final results with both problems:

eval: Between 250 and 260 iterations were observed to solve. eval1: Anywhere between 4,000 and 4,000,000 iterations were observed to solve.

Strengths and Weaknesses of the Algorithm

The bit progression method described above could easily trap the algorithm within a local optima if the distance metric were to differ. Perhaps a bit were flipped to the correct state, but the distance metric reports a lower value (for reasons unknown to us), and thus that bit is trapped in an invalid state and the algorithm never finds the solution.

In contrast, assuming the distance metric increases when the bit is flipped correctly, then the algorithm can quickly cut the search space down.

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string>

double eval(int*);

// Entry point
int main() {
    double result, oldResult;
    bool bitProgressive[100];
    int bit[100];
    int currentBitPosition = 0, oldBitPosition;
    int iteration = 0;
    int seed = time(NULL);

    // Init
    srand(seed);

    memset(bit, 0, 100 * sizeof(int));
    memset(bitProgressive, 1, 100 * sizeof(bool));

    // Bootstrap initial result
    result = eval(bit);
    while (result < 100.0f) {
        iteration += 1;

        // Check the flipped of current bit position
        bit[currentBitPosition] ^= 1;
        oldResult = result;
        result = eval(bit);

        // Revert if we went "downhill", set our current bit
        // to "non-progressive"
        if (result < oldResult) {
            bit[currentBitPosition] ^= 1;
            bitProgressive[currentBitPosition] = false;
        }

        // Find a new progressive bit position to flip
        oldBitPosition = currentBitPosition;
        do {
            currentBitPosition = rand() % 100;
        } while (!bitProgressive[currentBitPosition] ||
            oldBitPosition == currentBitPosition);
    }

    // Dump result
    printf("Result:\n");
}

```

```
    for (int i = 0; i < 100; i++)  
        printf("%i", bit[i]);  
    printf("\nTook %i iterations.\n", iteration);  
  
    return 0;  
}
```