CSC Problem Set 03 +X Report

Name: John Cheek
Email: jwcheek2@ncsu.edu
Section: 001
Collaborators: N/A

Summary:

For my +X percent for Problem Set 03 I compared three different algorithms to solve the linear assignment problem and compared their results. I compared the performance of a simulated annealing algorithm, a hill climbing algorithm (both based on pseudocode from the lecture slides) and the random search algorithm provided in the initial ConfigurationSolver java file. Of these three algorithms, which one was the 'best' heavily depends on what performance measure you are using. If the number of iterations is a concern, simulated annealing would be the most effective as it was the most efficient way of getting to a good solution quickly. If you simply need to solve the problem quickly, random search may be useful, though not ideal as the large number of iterations is very costly.

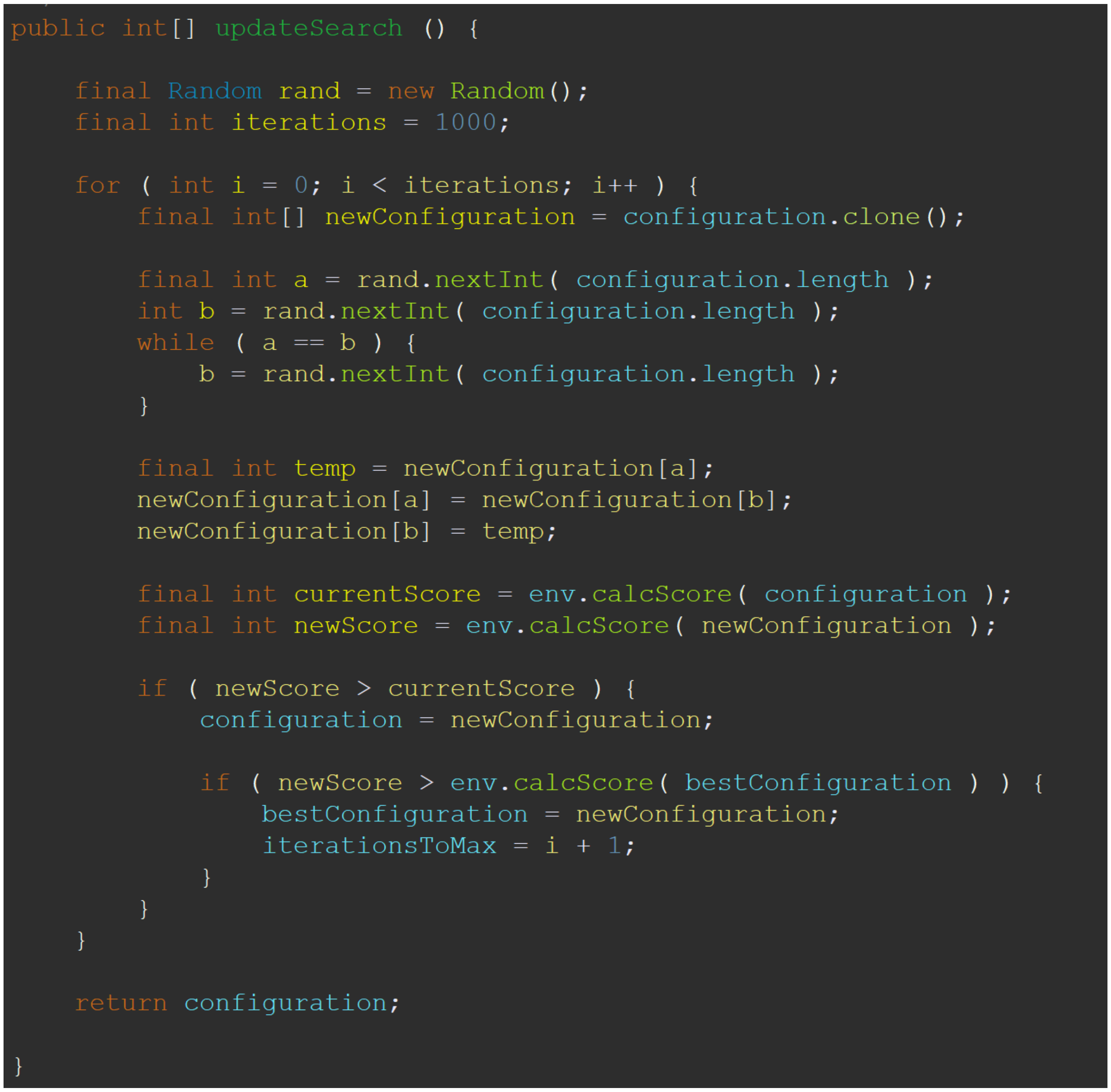
"+X" Concept:

I really enjoyed learning about the different naturally inspired algorithms to solve the linear assignment problem and wanted to apply them. For my initial approach I used simulated annealing which seemed to work pretty well but I wanted to compare it to some additional algorithms. I initially wanted to try genetic algorithms as an additional approach but I couldn't quite figure out how to best implement it so I decided to use hill climbing instead. I also figured that, while other biologically inspired algorithms may perform about the same as simulated annealing, the hill climbing algorithm would objectively perform worse so there would be a greater point of comparison and the analysis would be a little more interesting. Since it was provided, I figured I might as well compare these two approaches to the random search algorithm as well.

I also needed a way to measure the effectiveness of each algorithm. For this purpose, I decided it would be good to keep track of how many iterations it took to reach the best configuration. This was done by simply adding a field to the ConfigurationSolver class called 'iterationsToMax' and updating that field to the current iteration whenever the best configuration was updated. I also added a bit of code to the environment and simulation classes for the test to access this new field. The test will then calculate the average iterations to max throughout all of the trials.

Technical Implementation of +X:

Hill climbing:



```java
public int[] updateSearch () {

    final Random rand = new Random();
    final int iterations = 1000;

    for ( int i = 0; i < iterations; i++ ) {
        final int[] newConfiguration = configuration.clone();

        final int a = rand.nextInt( configuration.length );
        int b = rand.nextInt( configuration.length );
        while ( a == b ) {
            b = rand.nextInt( configuration.length );
        }

        final int temp = newConfiguration[a];
        newConfiguration[a] = newConfiguration[b];
        newConfiguration[b] = temp;

        final int currentScore = env.calcScore( configuration );
        final int newScore = env.calcScore( newConfiguration );

        if ( newScore > currentScore ) {
            configuration = newConfiguration;

            if ( newScore > env.calcScore( bestConfiguration ) ) {
                bestConfiguration = newConfiguration;
                iterationsToMax = i + 1;
            }
        }
    }

    return configuration;

}
```

My implementation of the hill climbing algorithm simply swaps two randomly chosen integers from the configuration. If this new configuration is greater than the previous one it will move to the new one (only if the new configuration is better). It will then update the best configuration and iterationsToMax accordingly.

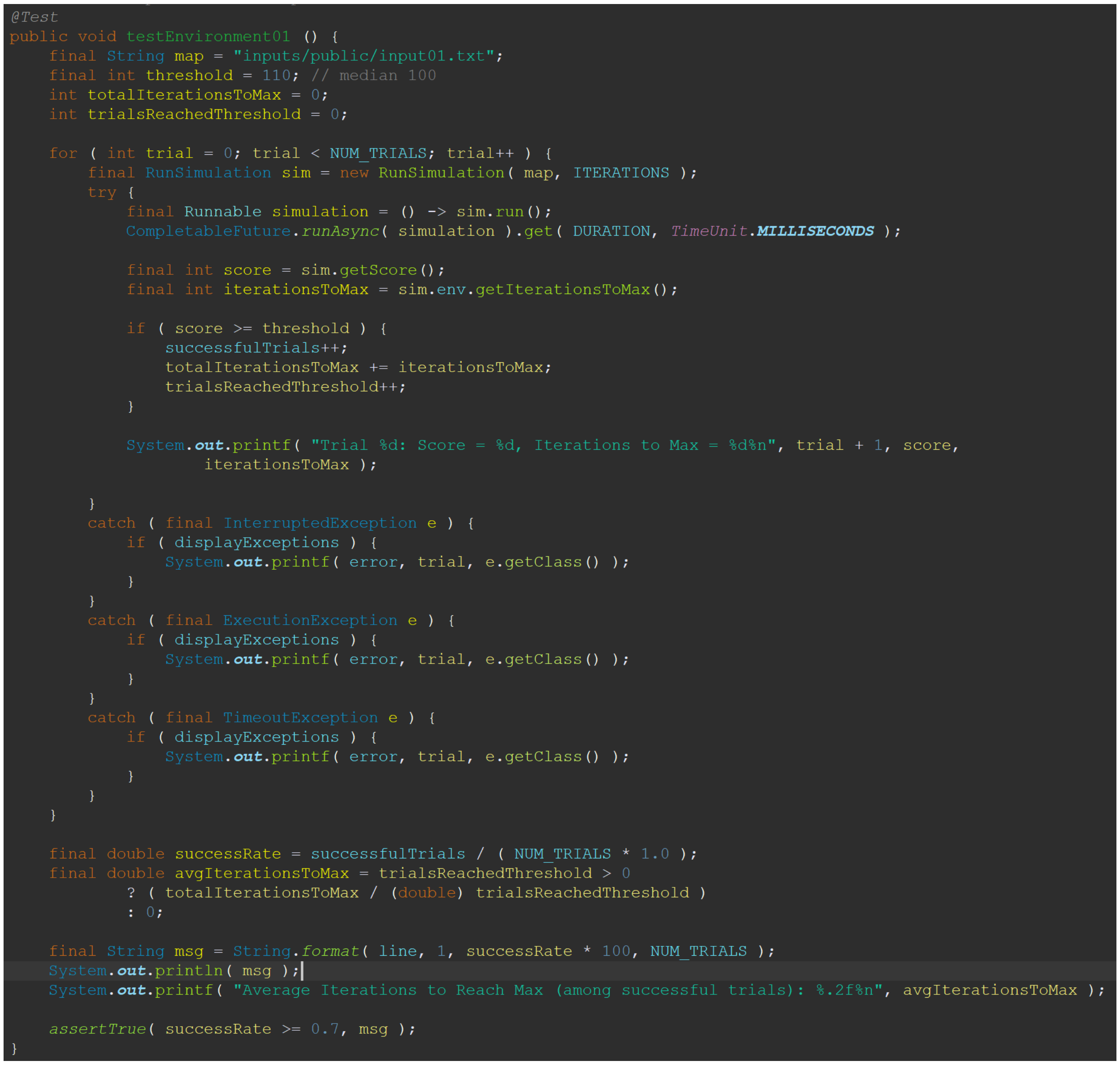
Simulated Annealing:

```
for ( int i = 0; i < 1000; i++ ) {

    final int temp = (int) schedule( i );

    if ( temp == 0 ) {

        return configuration;
    }

    final int[] newConfiguration = new int[configuration.length];

    for ( int j = 0; j < configuration.length; j++ ) {
        newConfiguration[j] = configuration[j];
    }

    final Random rand = new Random();
    final int swapIndex = rand.nextInt( configuration.length );

    newConfiguration[0] = configuration[swapIndex];
    newConfiguration[swapIndex] = configuration[0];

    if ( env.calcScore( newConfiguration ) > env.calcScore( bestConfiguration ) ) {
        this.bestConfiguration = newConfiguration;
    }

    final double E = env.calcScore( newConfiguration ) - env.calcScore( configuration );

    if ( E > 0 ) {
        this.configuration = newConfiguration;
    }
    else {
        final double prob = Math.exp( ( E ) / temp );
        if ( Math.random() < prob ) {
            this.configuration = newConfiguration;
        }
    }
}
return this.configuration;
```

```
public double schedule ( final int t ) {
    final double T0 = 50;
    final double c = 0.02;

    final double P = T0 * Math.exp( -c * t );

    if ( P < 0.005 ) {
        return 0;
    }
    else {
        return P;
    }

}
```

This is my solution to the initial problem set. It implements maximizing simulated annealing based on the pseudocode from the lecture slides and a schedule function with exponential temperature decay, an initial temperature of 50 and a decay rate of 0.02.

Modified Test Case:



```java
@Test
public void testEnvironment01 () {
    final String map = "inputs/public/input01.txt";
    final int threshold = 110; // median 100
    int totalIterationsToMax = 0;
    int trialsReachedThreshold = 0;

    for ( int trial = 0; trial < NUM_TRIALS; trial++ ) {
        final RunSimulation sim = new RunSimulation( map, ITERATIONS );
        try {
            final Runnable simulation = () -> sim.run();
            CompletableFuture.runAsync( simulation ).get( DURATION, TimeUnit.MILLISECONDS );

            final int score = sim.getScore();
            final int iterationsToMax = sim.env.getIterationsToMax();

            if ( score >= threshold ) {
                successfulTrials++;
                totalIterationsToMax += iterationsToMax;
                trialsReachedThreshold++;
            }

            System.out.printf( "Trial %d: Score = %d, Iterations to Max = %d%n", trial + 1, score,
                    iterationsToMax );

        }
        catch ( final InterruptedException e ) {
            if ( displayExceptions ) {
                System.out.printf( error, trial, e.getClass() );
            }
        }
        catch ( final ExecutionException e ) {
            if ( displayExceptions ) {
                System.out.printf( error, trial, e.getClass() );
            }
        }
        catch ( final TimeoutException e ) {
            if ( displayExceptions ) {
                System.out.printf( error, trial, e.getClass() );
            }
        }
    }

    final double successRate = successfulTrials / ( NUM_TRIALS * 1.0 );
    final double avgIterationsToMax = trialsReachedThreshold > 0
            ? ( totalIterationsToMax / (double) trialsReachedThreshold )
            : 0;

    final String msg = String.format( line, 1, successRate * 100, NUM_TRIALS );
    System.out.println( msg );
    System.out.printf( "Average Iterations to Reach Max (among successful trials): %.2f%n", avgIterationsToMax );

    assertTrue( successRate >= 0.7, msg );
}
```

This is my modification of the original test case. It calls the environment's getIterationsToMax() method (which returns the iterationsToMax field in the ConfigurationSolver class) every iteration to calculate the average number of iterations to max in that specific trial. It then prints that value at the end of the test.

Evaluation and Results:

Below are tables recording the results for each algorithm. The table contains the test number, the runtime of each test (in seconds), the average number of iterations it took to reach the best solution, and the test success rate.

Hill Climbing:

Test:	**1**	**2**	**3**	**4**	**5**
Runtime (s):	5	7	9	8	8
Average Iterations:	9.84	13.36	88.42	99.53	83.23
Test Success Rate:	100	100	100	100	100

Simulated Annealing:

Test:	**1**	**2**	**3**	**4**	**5**
Runtime (s):	4	5	5	6	4
Average Iterations:	70.76	18.46	61.51	67.70	64.14
Test Success Rate:	100	100	100	100	100

Random Search:

Test:	**1**	**2**	**3**	**4**	**5**
Runtime (s):	1	2	2	3	3
Average Iterations:	116.78	123.02	504.00	473.81	482.50
Test Success Rate:	100	100	100	100	100

These results were very interesting. I expected random search to be the worst, hill climbing to be better but not pass all the trials (in the event it got stuck at a local max), and for simulated annealing to be the best. HIll climbing had the longest run times and some of the later trials took more iterations. Simulated annealing had shorter runtimes (about 5s for all trials) and consistently took the fewest iterations.

Random search was the most surprising. I knew it worked, but I was surprised to see it had the shortest runtimes (about 3s for all trials) although it took many iterations to reach the max. The large number of iterations makes sense though, as this algorithm simply needs to randomize a list instead of going through 1000 iterations (like simulated annealing does).