760-Heuristics-Assignment

Leo Mooney

March 2023

1 Question 1

Neighbors of the current solution can be found by swapping one pot in any crucible with a pot in any other crucible. This can be formally defined as following:

$N(\mathbf{x}) = \{\mathbf{y}(\mathbf{x}, k, l, m, n), k = 1, 2, 3, ..., 16, l = k + 1, k + 2, k + 3, ...17, m = 1, 2, 3, n = 1, 2, 3\}$ where

$$\mathbf{y}(\mathbf{x}, k, l, m, n) = (y_{1,1}, y_{1,2}, y_{1,3}; y_{2,1}, y_{2,2}, y_{2,3}; ...y_{17,3}), y_{c,j} = \begin{cases} x_{l,n} & \text{if } c = k, j = m \\ x_{k,m} & \text{if } c = l, j = n \\ x_{c,j} & \text{otherwise} \end{cases}$$

2 Question 2

The intermediate values for each crucible will be the value of function $g()$ of that crucible. The intermediate values will only store the values of the crucibles in the current configuration x, and will be updated when x is updated.

Let $h(x_c) = g(\overline{Al}[x_{c,avg}], \overline{Fe}[x_{c,avg}], \overline{Si}[x_{c,avg}])$ where

$$\overline{Al}[x_{c,avg}] = \frac{Al[x_{c,1}] + Al[x_{c,2}] + Al[x_{c,3}]}{3}$$

$$\overline{Fe}[x_{c,avg}] = \frac{Fe[x_{c,1}] + Fe[x_{c,2}] + Fe[x_{c,3}]}{3}$$

$$\overline{Si}[x_{c,avg}] = \frac{Si[x_{c,1}] + Si[x_{c,2}] + Si[x_{c,3}]}{3}$$

Algorithm 1 Sweep x

Let S contain all possible solutions of x
Let x, $x \in S$, be some initial configuration
Let x^* be some configuration that maximizes f(x)
Let I be the intermediate values
$I_i := h(x_i) \forall i \in c$
while not stopped **do**
  for $k = 1$ to 17 **do**
    for $m = 1$ to 3 **do**
      for $l = k + 1$ to 17 **do**
        for $n = 1$ to 3 **do**
          $y = y(x, k, l, m, n)$
          Let $\Delta = h(y_k) + h(y_l) - I_k - I_l$
          if $\Delta > 0$ **then**
            x := y
            $I_k := h(y_k)$
            $I_l := h(y_l)$
          else if $\Delta \leq 0 \forall y \in N(x)$ **then**
            Stop
          end if
        end for
      end for
    end for
  end for
end while
$x^* := x$

3 Question 3

See Appendix A for all code and plots.

3.1 Question 3E

Best solution found using repeated next ascents with $n = 200$.

```
 1 [27  4 45 ] 99.261 %Al, 0.274 %Fe, 0.110 %Si, $41.53, spread = 41
 2 [ 8 13 37 ] 99.508 %Al, 0.154 %Fe, 0.153 %Si, $48.71, spread = 29
 3 [50 40 30 ] 99.504 %Al, 0.143 %Fe, 0.238 %Si, $48.71, spread = 20
 4 [17 51 15 ] 99.268 %Al, 0.162 %Fe, 0.386 %Si, $41.53, spread = 36
 5 [24 23 46 ] 99.512 %Al, 0.053 %Fe, 0.180 %Si, $48.71, spread = 23
 6 [28 12 32 ] 99.358 %Al, 0.096 %Fe, 0.348 %Si, $44.53, spread = 20
 7 [18  3 42 ] 99.760 %Al, 0.040 %Fe, 0.139 %Si, $57.35, spread = 39
 8 [19 41 20 ] 99.262 %Al, 0.212 %Fe, 0.249 %Si, $41.53, spread = 22
 9 [11 25  5 ] 99.353 %Al, 0.185 %Fe, 0.325 %Si, $44.53, spread = 20
10 [ 9  6 21 ] 99.504 %Al, 0.139 %Fe, 0.253 %Si, $48.71, spread = 15
11 [ 7 36 26 ] 99.506 %Al, 0.098 %Fe, 0.268 %Si, $48.71, spread = 29
12 [47 39  2 ] 99.503 %Al, 0.114 %Fe, 0.270 %Si, $48.71, spread = 45
13 [22 33 10 ] 99.356 %Al, 0.187 %Fe, 0.219 %Si, $44.53, spread = 23
14 [34 29 38 ] 99.361 %Al, 0.209 %Fe, 0.241 %Si, $44.53, spread =  9
15 [14 43 49 ] 99.254 %Al, 0.224 %Fe, 0.307 %Si, $41.53, spread = 35
16 [ 1 35 31 ] 99.350 %Al, 0.075 %Fe, 0.336 %Si, $44.53, spread = 34
17 [48 44 16 ] 99.515 %Al, 0.143 %Fe, 0.241 %Si, $48.71, spread = 32
                                           Sum = $787.09, MxSprd = 45
```

Best solution found using repeated steepest ascents with $n = 200$.

```
 1 [ 1 48 15 ] 99.256 %Al, 0.121 %Fe, 0.300 %Si, $41.53, spread = 47
 2 [29 36 10 ] 99.369 %Al, 0.216 %Fe, 0.272 %Si, $44.53, spread = 26
 3 [33  6 34 ] 99.268 %Al, 0.253 %Fe, 0.272 %Si, $41.53, spread = 28
 4 [42 16 49 ] 99.502 %Al, 0.161 %Fe, 0.163 %Si, $48.71, spread = 33
 5 [43 24 39 ] 99.508 %Al, 0.067 %Fe, 0.243 %Si, $48.71, spread = 19
 6 [47  8  5 ] 99.366 %Al, 0.188 %Fe, 0.317 %Si, $44.53, spread = 42
 7 [ 4 26 19 ] 99.253 %Al, 0.243 %Fe, 0.283 %Si, $41.53, spread = 22
 8 [31 28 46 ] 99.365 %Al, 0.139 %Fe, 0.196 %Si, $44.53, spread = 18
 9 [ 3 11 23 ] 99.502 %Al, 0.058 %Fe, 0.246 %Si, $48.71, spread = 20
10 [32 27 35 ] 99.501 %Al, 0.087 %Fe, 0.224 %Si, $48.71, spread =  8
11 [22 40 51 ] 99.353 %Al, 0.122 %Fe, 0.349 %Si, $44.53, spread = 29
12 [13 38  2 ] 99.504 %Al, 0.177 %Fe, 0.160 %Si, $48.71, spread = 36
13 [50 25 37 ] 99.500 %Al, 0.188 %Fe, 0.231 %Si, $48.71, spread = 25
14 [20 30 17 ] 99.522 %Al, 0.111 %Fe, 0.277 %Si, $48.71, spread = 13
15 [ 9 18 44 ] 99.750 %Al, 0.037 %Fe, 0.099 %Si, $57.35, spread = 35
16 [45 41 12 ] 99.258 %Al, 0.171 %Fe, 0.284 %Si, $41.53, spread = 33
17 [21  7 14 ] 99.360 %Al, 0.168 %Fe, 0.347 %Si, $44.53, spread = 14
                                           Sum = $787.09, MxSprd = 47
```

4 Quesion 4

There does not seem to be a significant difference in the objective value for next and steepest ascent. Steepest ascent also took far longer to converge on each local optimum. This means that fastest ascent is better than steepest ascent; however, this conclusion is from limited information so it not conclusive evidence.

5 Question 5

You would expect the problem's objective function to have lots of plateus because the objective function is not continuous. This means there will be lots of cases where two pots of similar quality are swapped and the grade, and thus value, will remain constant. It will also be common that two pots of different qualities will swap, causing their grades to also swap. In this case the value will also remain constant.

The best outcome is when the crucible is just above the min/max requirements to enter the grade boundary. This is because when sitting too far above the grade boundary there are wasted resources (i.e. the quality doesn't need to be that good). The proposed function will add a non-linear gradient to the existing value function to encourage sitting on the grade boundary. This non-linear gradient will take the form of a quadratic with a local minimum at the current grade boundary. The maximum value this quadratic will take will be 1 at the next grade boundary and the minimum 0. This can be formally defined as follows

$$g'(\overline{Al}, \overline{Fe}, \overline{Si}) = g(\overline{Al}, \overline{Fe}, \overline{Si}) + \frac{(p-r)^2}{(q-r)^2}$$

Where
p is which of $\overline{Al}, \overline{Fe}, \overline{Si}$ is closest to the current grade min/max requirement.
r is the current grade min/max requirement for p.
q is the next highest grade min/max requirement for p.

The effect this will have is that there will be a gradient encouraging neighbors to be closer to the next grade boundary. Furthermore, neighbors that are already closer to the next grade boundary will be further encouraged than those who are far away from the next grade boundary.

Let us consider a simplified example where the crucible only has one relevant property, Al. This means that Si and Fe means are already the minimum then can be. If $Al > 95$ then the value is \$10. If $Al > 97$ then the value is \$20. Now let us consider solution x with two crucibles x_k, x_l, and a neighbor y with two crucibles y_k, y_l.

$Al[x_{k,1}] = 95.3, Al[x_{k,2}] = 95.2, Al[x_{k,3}] = 96.5, \overline{Al}[x_k] = 95.67$
$Al[x_{l,1}] = 96.8, Al[x_{l,2}] = 96.9, Al[x_{l,3}] = 95.4, \overline{Al}[x_l] = 96.37$
$Al[y_{k,1}] = 95.3, Al[y_{k,2}] = 95.2, Al[y_{k,3}] = 95.4, \overline{Al}[y_k] = 95.30$
$Al[y_{l,1}] = 96.8, Al[y_{l,2}] = 96.9, Al[y_{l,3}] = 96.5, \overline{Al}[y_l] = 96.73$

In this example using the original grading function there would be no difference between the objective value for x and y so the neighbor would not be accepted in standard next/steepest ascent. However, accepting the new solution would likely allow a better solution to be found, as crucible l is closer to the \$97 boundary while k is closer to the \$95 boundary, which it will not drop below without good reason due to the sharp change in grading function $g()$ at this boundary. On a next iteration it will be easier for pot l to exceed the grade boundary giving it a higher value if this swap is made.

Using the modified $g'()$ The crucible values are as follows
$x_k = \$10.112225$ $x_l = \$10.469225$ $y_k = \$10.022500$ $y_l = \$10.748225$
This results in a change of objective value of 0.189275, meaning the better solution will be accepted. The greatest magnitude this gradient value can take is 1, as it is stil desirable to maintain a steep gradient when changing grades.

Note: This new objective function with next ascent ended up working better than simulated annealing and was used to find my best solutions.

6 Question 6

$$g''(\overline{Al}, \overline{Fe}, \overline{Si}, x_{c1}, x_{c2}, x_{c3}, s) = \begin{cases} g(\overline{Al}, \overline{Fe}, \overline{Si}) - 20 * (s_c - s) & \text{if } s_c > s \\ g(\overline{Al}, \overline{Fe}, \overline{Si}) & \text{otherwise} \end{cases}$$

where $s_c = \max(x_{c1}, x_{c2}, x_{c3}) - \min(x_{c1}, x_{c2}, x_{c3})$

6.1 Task 6

See Appendix A for code.

6.1.1 Max Spread = 6

```
 1 [38 34 40 ] 99.371 %Al, 0.241 %Fe, 0.163 %Si, $44.53, spread =  6
 2 [17 13 19 ] 99.270 %Al, 0.297 %Fe, 0.234 %Si, $41.53, spread =  6
 3 [35 36 39 ] 99.615 %Al, 0.043 %Fe, 0.196 %Si, $48.71, spread =  4
 4 [ 6  8 10 ] 99.352 %Al, 0.257 %Fe, 0.275 %Si, $44.53, spread =  4
 5 [30 32 27 ] 99.558 %Al, 0.093 %Fe, 0.170 %Si, $48.71, spread =  5
 6 [37 42 41 ] 99.501 %Al, 0.115 %Fe, 0.198 %Si, $48.71, spread =  5
 7 [24 22 21 ] 99.532 %Al, 0.037 %Fe, 0.276 %Si, $48.71, spread =  3
 8 [33 29 28 ] 99.256 %Al, 0.151 %Fe, 0.303 %Si, $41.53, spread =  5
 9 [11 14  9 ] 99.381 %Al, 0.080 %Fe, 0.297 %Si, $44.53, spread =  5
```

```
10 [ 5  7  3 ] 99.512 %Al, 0.188 %Fe, 0.212 %Si, $48.71, spread =  4
11 [46 47 43 ] 99.359 %Al, 0.172 %Fe, 0.315 %Si, $44.53, spread =  4
12 [26 25 31 ] 99.393 %Al, 0.138 %Fe, 0.325 %Si, $44.53, spread =  6
13 [49 48 44 ] 99.511 %Al, 0.169 %Fe, 0.145 %Si, $48.71, spread =  5
14 [20 18 23 ] 99.544 %Al, 0.057 %Fe, 0.220 %Si, $48.71, spread =  5
15 [15 16 12 ] 99.369 %Al, 0.127 %Fe, 0.302 %Si, $44.53, spread =  4
16 [ 2  4  1 ] 99.356 %Al, 0.144 %Fe, 0.280 %Si, $44.53, spread =  3
17 [50 51 45 ] 99.256 %Al, 0.198 %Fe, 0.351 %Si, $41.53, spread =  6
                                           Sum = $777.27, MxSprd =  6
```

6.1.2 Max Spread = 8

```
 1 [15 13 11 ] 99.255 %Al, 0.157 %Fe, 0.277 %Si, $41.53, spread =  4
 2 [ 2  1  4 ] 99.356 %Al, 0.144 %Fe, 0.280 %Si, $44.53, spread =  3
 3 [46 49 51 ] 99.399 %Al, 0.152 %Fe, 0.259 %Si, $44.53, spread =  5
 4 [26 19 24 ] 99.355 %Al, 0.140 %Fe, 0.305 %Si, $44.53, spread =  7
 5 [32 29 36 ] 99.509 %Al, 0.087 %Fe, 0.197 %Si, $48.71, spread =  7
 6 [34 35 33 ] 99.358 %Al, 0.156 %Fe, 0.278 %Si, $44.53, spread =  2
 7 [ 8  3 10 ] 99.527 %Al, 0.176 %Fe, 0.194 %Si, $48.71, spread =  7
 8 [48 50 44 ] 99.544 %Al, 0.129 %Fe, 0.241 %Si, $48.71, spread =  6
 9 [14  6  7 ] 99.253 %Al, 0.249 %Fe, 0.360 %Si, $41.53, spread =  8
10 [16 18 17 ] 99.542 %Al, 0.132 %Fe, 0.258 %Si, $48.71, spread =  2
11 [30 27 23 ] 99.505 %Al, 0.105 %Fe, 0.199 %Si, $48.71, spread =  7
12 [20 22 21 ] 99.516 %Al, 0.060 %Fe, 0.277 %Si, $48.71, spread =  2
13 [43 45 38 ] 99.350 %Al, 0.226 %Fe, 0.125 %Si, $44.53, spread =  7
14 [ 9  5 12 ] 99.510 %Al, 0.106 %Fe, 0.258 %Si, $48.71, spread =  7
15 [40 39 47 ] 99.378 %Al, 0.169 %Fe, 0.269 %Si, $44.53, spread =  8
16 [28 25 31 ] 99.278 %Al, 0.204 %Fe, 0.285 %Si, $41.53, spread =  6
17 [42 41 37 ] 99.501 %Al, 0.115 %Fe, 0.198 %Si, $48.71, spread =  5
                                           Sum = $781.45, MxSprd =  8
```

6.1.3 Max Spread = 11

```
 1 [40 49 51 ] 99.301 %Al, 0.205 %Fe, 0.295 %Si, $41.53, spread = 11
 2 [35 33 34 ] 99.358 %Al, 0.156 %Fe, 0.278 %Si, $44.53, spread =  2
 3 [ 5  3  7 ] 99.512 %Al, 0.188 %Fe, 0.212 %Si, $48.71, spread =  4
 4 [29 31 28 ] 99.257 %Al, 0.160 %Fe, 0.310 %Si, $41.53, spread =  3
 5 [ 4  6  9 ] 99.386 %Al, 0.217 %Fe, 0.204 %Si, $44.53, spread =  5
 6 [36 32 26 ] 99.524 %Al, 0.048 %Fe, 0.266 %Si, $48.71, spread = 10
 7 [39 44 47 ] 99.500 %Al, 0.124 %Fe, 0.270 %Si, $48.71, spread =  8
 8 [25 18 16 ] 99.527 %Al, 0.153 %Fe, 0.256 %Si, $48.71, spread =  9
 9 [41 50 48 ] 99.351 %Al, 0.151 %Fe, 0.329 %Si, $44.53, spread =  9
10 [14 22 24 ] 99.362 %Al, 0.083 %Fe, 0.335 %Si, $44.53, spread = 10
11 [13 23 19 ] 99.251 %Al, 0.258 %Fe, 0.213 %Si, $41.53, spread = 10
12 [30 27 20 ] 99.530 %Al, 0.104 %Fe, 0.212 %Si, $48.71, spread = 10
13 [46 42 45 ] 99.502 %Al, 0.140 %Fe, 0.056 %Si, $48.71, spread =  4
```

```
14 [21 10 17 ] 99.369 %Al, 0.239 %Fe, 0.344 %Si, $44.53, spread = 11
15 [37 43 38 ] 99.519 %Al, 0.169 %Fe, 0.143 %Si, $48.71, spread =  6
16 [11 12 15 ] 99.361 %Al, 0.068 %Fe, 0.309 %Si, $44.53, spread =  4
17 [ 8  1  2 ] 99.526 %Al, 0.042 %Fe, 0.230 %Si, $48.71, spread =  7
                                            Sum = $781.45, MxSprd = 11
```

7 My Best Solutions

These were found by using my modified objective function from Q5 in C with next ascent. These solutions took around 1500 iterations to converge upon with my updated objective function. This gave slightly better results than simulated annealing in C. I am happy to provide the code if required for the competition, just email me.

7.1 No Max Spread

```
 1 [35 23 19 ] 99.356 %Al, 0.172 %Fe, 0.239 %Si, $44.53, spread = 16
 2 [ 5 49 26 ] 99.351 %Al, 0.188 %Fe, 0.316 %Si, $44.53, spread = 44
 3 [ 6 43  7 ] 99.350 %Al, 0.235 %Fe, 0.300 %Si, $44.53, spread = 37
 4 [24 33  1 ] 99.350 %Al, 0.056 %Fe, 0.283 %Si, $44.53, spread = 32
 5 [12 48 45 ] 99.352 %Al, 0.166 %Fe, 0.235 %Si, $44.53, spread = 36
 6 [13 39 37 ] 99.500 %Al, 0.154 %Fe, 0.170 %Si, $48.71, spread = 26
 7 [17 20 34 ] 99.351 %Al, 0.201 %Fe, 0.308 %Si, $44.53, spread = 17
 8 [29 38 41 ] 99.355 %Al, 0.154 %Fe, 0.261 %Si, $44.53, spread = 12
 9 [11  2 16 ] 99.500 %Al, 0.087 %Fe, 0.272 %Si, $48.71, spread = 14
10 [47  3 15 ] 99.351 %Al, 0.172 %Fe, 0.287 %Si, $44.53, spread = 44
11 [28 50 22 ] 99.350 %Al, 0.126 %Fe, 0.329 %Si, $44.53, spread = 28
12 [51  9 36 ] 99.517 %Al, 0.070 %Fe, 0.264 %Si, $48.71, spread = 42
13 [27 25  4 ] 99.350 %Al, 0.253 %Fe, 0.204 %Si, $44.53, spread = 23
14 [14 40 46 ] 99.351 %Al, 0.173 %Fe, 0.232 %Si, $44.53, spread = 32
15 [30  8 10 ] 99.500 %Al, 0.167 %Fe, 0.227 %Si, $48.71, spread = 22
16 [21 32 31 ] 99.501 %Al, 0.089 %Fe, 0.201 %Si, $48.71, spread = 11
17 [44 42 18 ] 99.750 %Al, 0.044 %Fe, 0.136 %Si, $57.35, spread = 26
                                            Sum = $790.73, MxSprd = 44
```

7.2 Max Spread = 6

```
 1 [44,49,48,] 99.511 %Al, 0.169 %Fe, 0.145 %Si, $48.71, spread=5
 2 [18,19,21,] 99.512 %Al, 0.163 %Fe, 0.187 %Si, $48.71, spread=3
 3 [43,47,46,] 99.359 %Al, 0.172 %Fe, 0.315 %Si, $44.53, spread=4
 4 [9,11,12,] 99.535 %Al, 0.015 %Fe, 0.265 %Si, $48.71, spread=3
 5 [27,24,22,] 99.502 %Al, 0.068 %Fe, 0.194 %Si, $48.71, spread=5
 6 [36,39,34,] 99.502 %Al, 0.139 %Fe, 0.173 %Si, $48.71, spread=5
 7 [25,23,28,] 99.267 %Al, 0.185 %Fe, 0.332 %Si, $41.53, spread=5
 8 [38,37,42,] 99.650 %Al, 0.118 %Fe, 0.067 %Si, $52.44, spread=5
```

```
 9 [51,50,45,] 99.256 %Al, 0.198 %Fe, 0.351 %Si, $41.53, spread=6
 10 [2,7,5,] 99.505 %Al, 0.182 %Fe, 0.208 %Si, $48.71, spread=5
 11 [29,26,31,] 99.372 %Al, 0.094 %Fe, 0.350 %Si, $44.53, spread=5
 12 [16,14,20,] 99.298 %Al, 0.164 %Fe, 0.379 %Si, $41.53, spread=6
 13 [33,32,30,] 99.541 %Al, 0.070 %Fe, 0.160 %Si, $48.71, spread=3
 14 [6,8,10,] 99.352 %Al, 0.257 %Fe, 0.275 %Si, $44.53, spread=4
 15 [3,1,4,] 99.363 %Al, 0.151 %Fe, 0.284 %Si, $44.53, spread=3
 16 [13,15,17,] 99.276 %Al, 0.220 %Fe, 0.261 %Si, $41.53, spread=4
 17 [35,40,41,] 99.335 %Al, 0.141 %Fe, 0.317 %Si, $41.53, spread=6
Sum = $779.18, MxSprd = 6
```

7.3 Max Spread = 8

```
 1 [8,1,6,] 99.358 %Al, 0.130 %Fe, 0.315 %Si, $44.53, spread=7
 2 [24,27,21,] 99.506 %Al, 0.091 %Fe, 0.245 %Si, $48.71, spread=6
 3 [26,31,29,] 99.372 %Al, 0.094 %Fe, 0.350 %Si, $44.53, spread=5
 4 [32,40,34,] 99.353 %Al, 0.199 %Fe, 0.210 %Si, $44.53, spread=8
 5 [46,44,41,] 99.507 %Al, 0.108 %Fe, 0.191 %Si, $48.71, spread=5
 6 [22,15,18,] 99.514 %Al, 0.064 %Fe, 0.164 %Si, $48.71, spread=7
 7 [17,13,20,] 99.359 %Al, 0.191 %Fe, 0.305 %Si, $44.53, spread=7
 8 [38,33,35,] 99.501 %Al, 0.105 %Fe, 0.167 %Si, $48.71, spread=5
 9 [47,39,43,] 99.370 %Al, 0.153 %Fe, 0.341 %Si, $44.53, spread=8
 10 [7,2,5,] 99.505 %Al, 0.182 %Fe, 0.208 %Si, $48.71, spread=5
 11 [51,45,50,] 99.256 %Al, 0.198 %Fe, 0.351 %Si, $41.53, spread=6
 12 [12,16,19,] 99.362 %Al, 0.204 %Fe, 0.275 %Si, $44.53, spread=7
 13 [11,9,14,] 99.381 %Al, 0.080 %Fe, 0.297 %Si, $44.53, spread=5
 14 [36,30,37,] 99.654 %Al, 0.098 %Fe, 0.129 %Si, $52.44, spread=7
 15 [10,3,4,] 99.357 %Al, 0.278 %Fe, 0.244 %Si, $44.53, spread=7
 16 [49,48,42,] 99.513 %Al, 0.147 %Fe, 0.139 %Si, $48.71, spread=7
 17 [28,25,23,] 99.267 %Al, 0.185 %Fe, 0.332 %Si, $41.53, spread=5
Sum = $784.00, MxSprd = 8
```

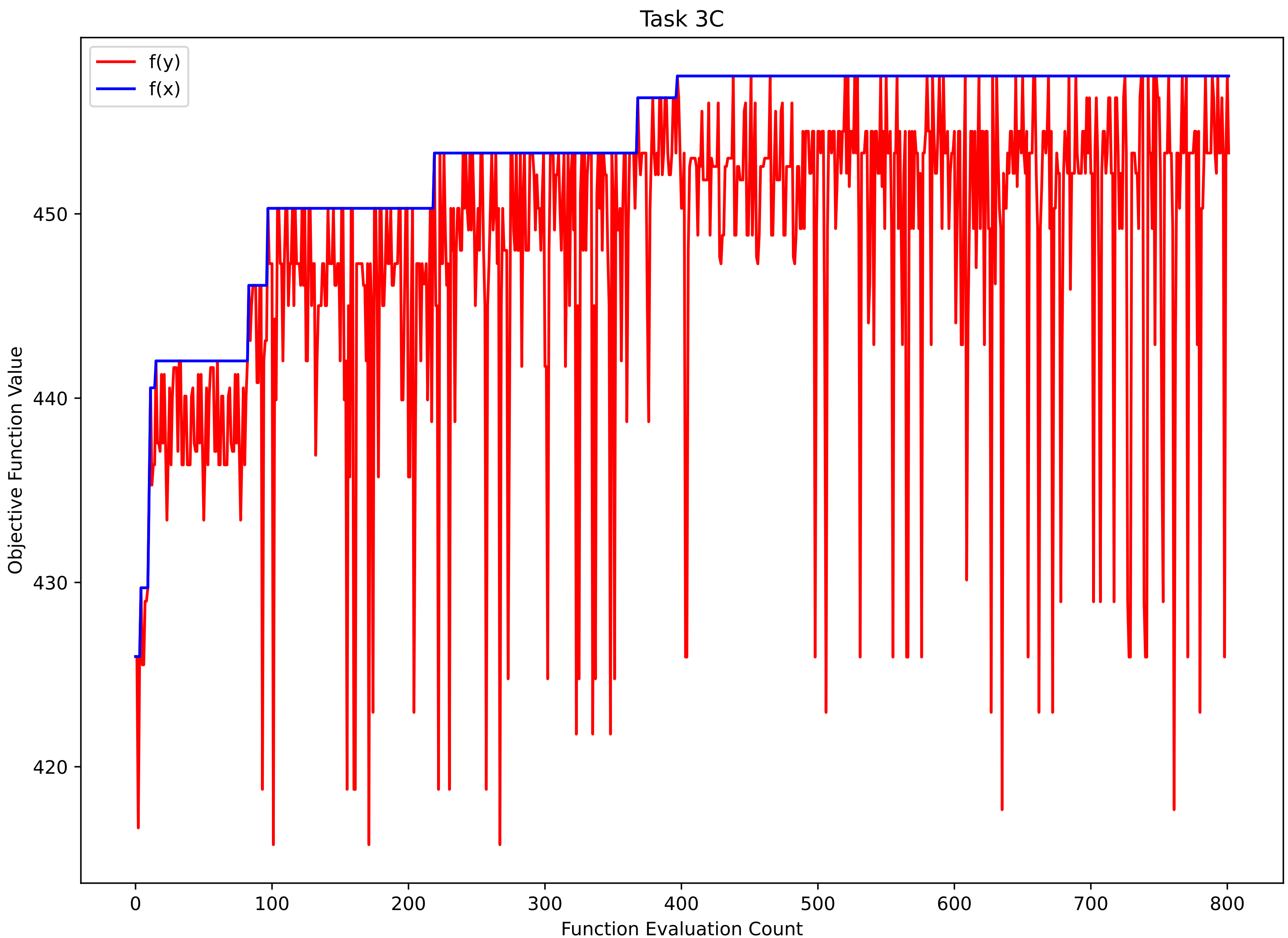
7.4 Max Spread = 11

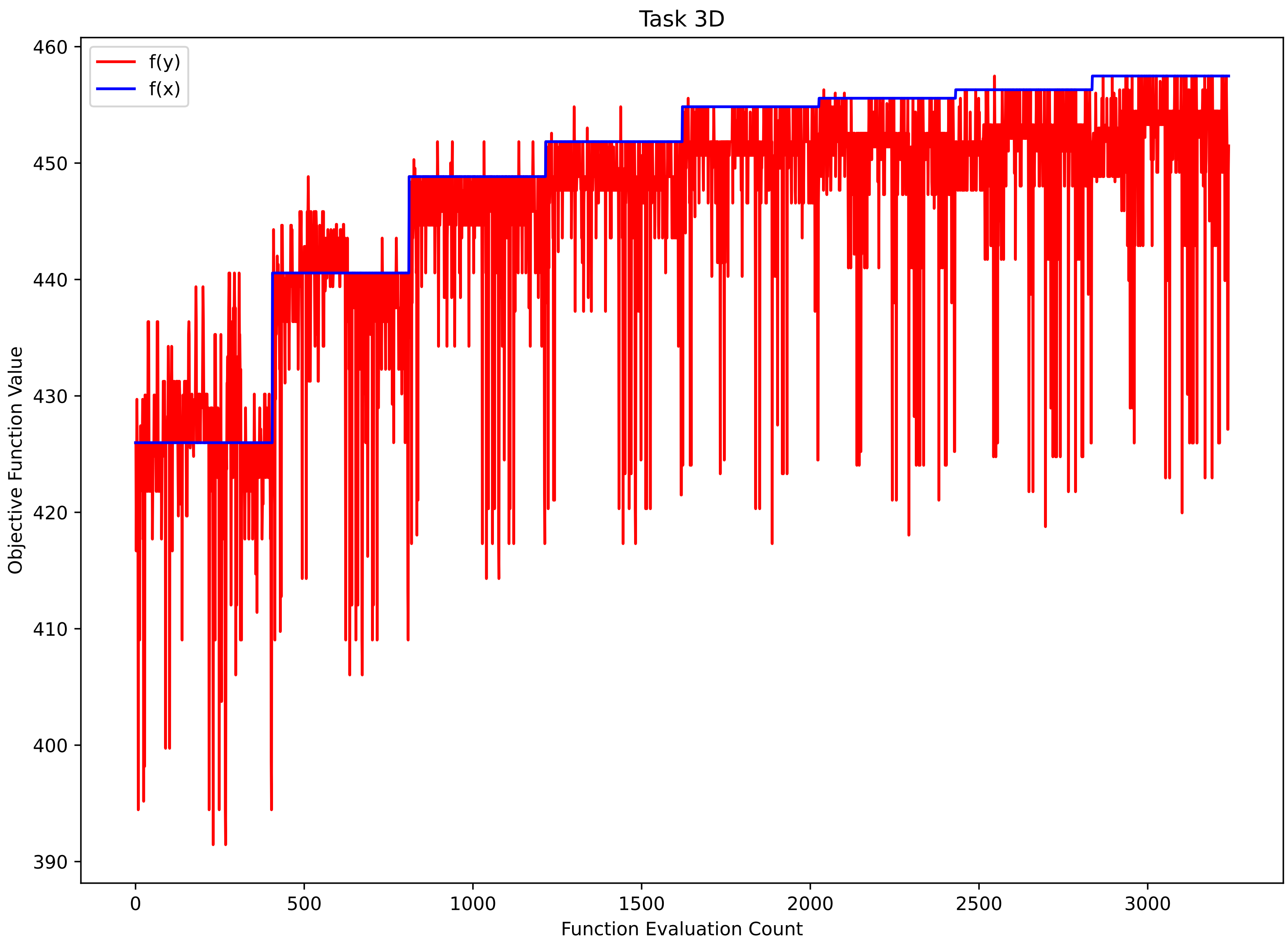
```
 1 [49,46,51,] 99.399 %Al, 0.152 %Fe, 0.259 %Si, $44.53, spread=5
 2 [26,37,32,] 99.514 %Al, 0.072 %Fe, 0.266 %Si, $48.71, spread=11
 3 [19,21,29,] 99.356 %Al, 0.204 %Fe, 0.263 %Si, $44.53, spread=10
 4 [20,30,23,] 99.503 %Al, 0.072 %Fe, 0.255 %Si, $48.71, spread=10
 5 [25,28,36,] 99.360 %Al, 0.186 %Fe, 0.275 %Si, $44.53, spread=11
 6 [2,1,4,] 99.356 %Al, 0.144 %Fe, 0.280 %Si, $44.53, spread=3
 7 [17,16,6,] 99.354 %Al, 0.238 %Fe, 0.341 %Si, $44.53, spread=11
 8 [34,35,33,] 99.358 %Al, 0.156 %Fe, 0.278 %Si, $44.53, spread=2
 9 [45,43,38,] 99.350 %Al, 0.226 %Fe, 0.125 %Si, $44.53, spread=7
 10 [10,3,13,] 99.353 %Al, 0.271 %Fe, 0.287 %Si, $44.53, spread=10
 11 [39,44,47,] 99.500 %Al, 0.124 %Fe, 0.270 %Si, $48.71, spread=8
 12 [7,9,5,] 99.504 %Al, 0.163 %Fe, 0.166 %Si, $48.71, spread=4
```

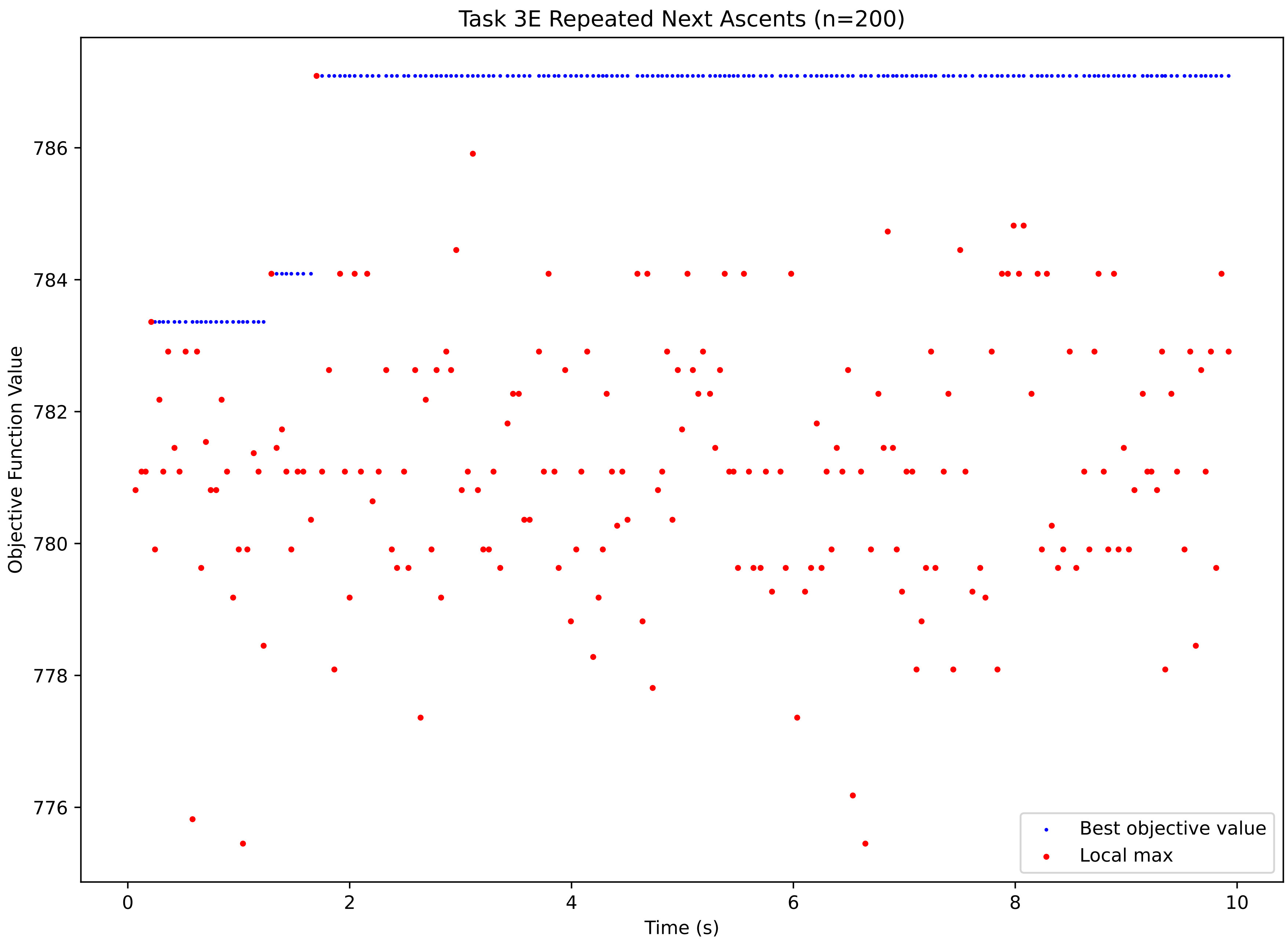
```
 13 [42,40,31,] 99.500 %Al, 0.131 %Fe, 0.120 %Si, $48.71, spread=11
 14 [11,8,14,] 99.375 %Al, 0.082 %Fe, 0.312 %Si, $44.53, spread=6
 15 [22,27,24,] 99.502 %Al, 0.068 %Fe, 0.194 %Si, $48.71, spread=5
 16 [15,12,18,] 99.502 %Al, 0.068 %Fe, 0.241 %Si, $48.71, spread=6
 17 [48,41,50,] 99.351 %Al, 0.151 %Fe, 0.329 %Si, $44.53, spread=9
Sum = $786.27, MxSprd = 11
```

8 Appendix A

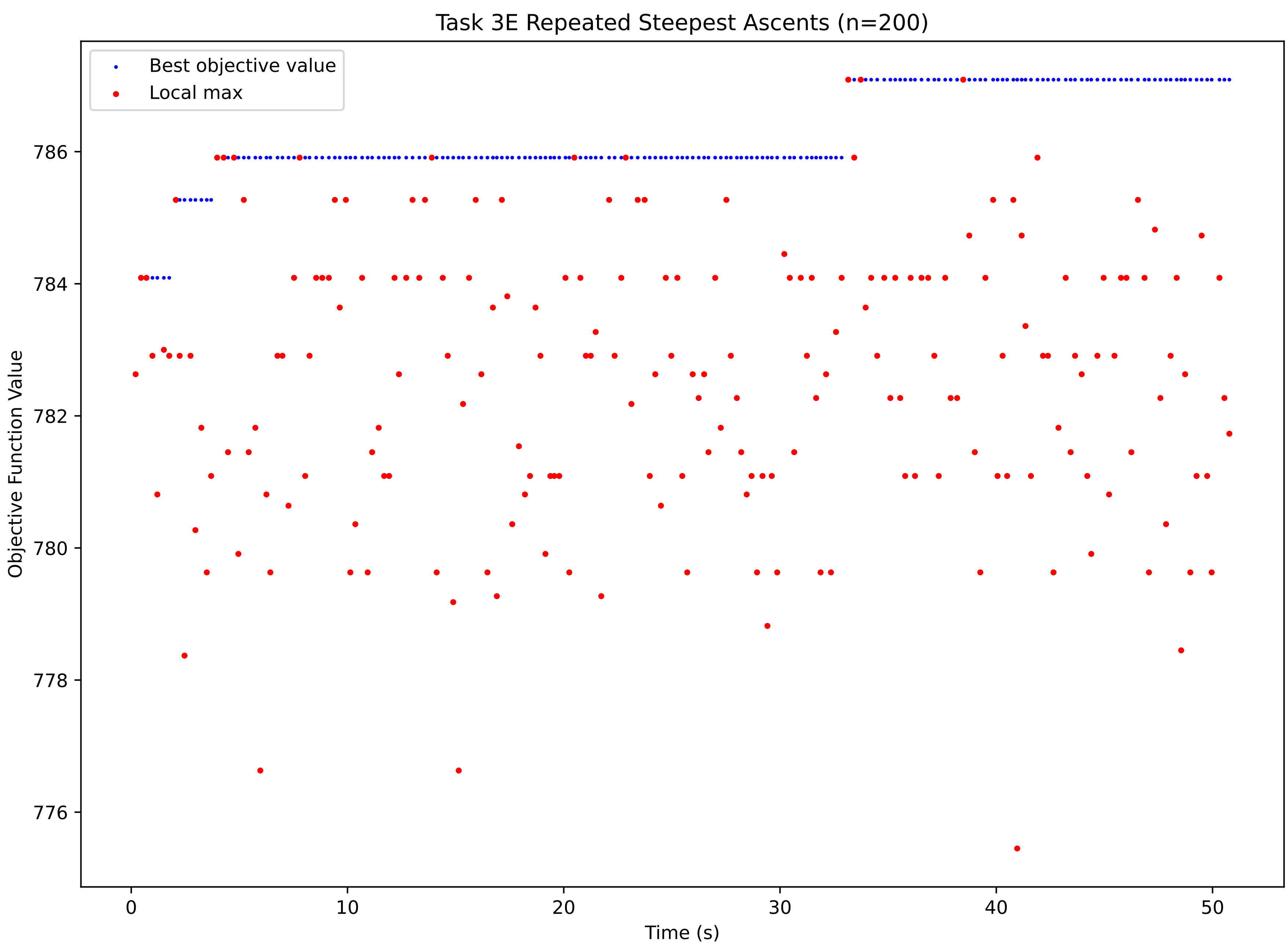
See overleaf


Task 3C
f(y)
f(x)
Objective Function Value
Function Evaluation Count
420
430
440
450
0
100
200
300
400
500
600
700
800


Task 3D
f(y)
f(x)
Objective Function Value
Function Evaluation Count
460
450
440
430
420
410
400
390
0
500
1000
1500
2000
2500
3000



Task 3E Repeated Next Ascents (n=200)
Objective Function Value
Time (s)
786
784
782
780
778
776
0
2
4
6
8
10
Best objective value
Local max



Task 3E Repeated Steepest Ascents (n=200)
Best objective value
Local max
786
784
782
780
778
776
Objective Function Value
0
10
20
30
40
50
Time (s)

```python
# (C) Andrew Mason 2023 - ENGSCI 760 Heuristics Assignment
# This code, and any code derived from this, may NOT be posted in any publicly accessible location
# Specifically, this code, and any derived versions of this code (including your assignment answers)
# must NOT be posted publically on Github, Gitlab or similar.

import numpy as np
import matplotlib.pyplot as plt
from enum import IntEnum
import time
import random

class Element(IntEnum):
    """The elements that we measure levels of in the Aluminium we produce"""
    Al = 0
    Fe = 1
    Si = 2

class LocalSearch():
    def __init__(self) -> None:
        self.load_default_problem()

    def load_default_problem(self) -> None:
        """Initialise the configuration parameters with default values"""
        self.no_crucibles=17
        self.no_pots=51
        self.pots_per_crucible=3
        # Initialise the percentage of Al (aluminium), Fe (iron) and Silicon (Si)
        self.pot_quality = np.array(
                           [ [99.136, 0.051, 0.497],
                             [99.733, 0.064, 0.138],
                             [99.755, 0.083, 0.149],
                             [99.198, 0.318, 0.206],
                             [99.297, 0.284, 0.33],
                             [99.23, 0.327, 0.393],
                             [99.485, 0.197, 0.156],
                             [99.709, 0.011, 0.056],
                             [99.729, 0.007, 0.012],
                             [99.118, 0.434, 0.377],
                             [99.372, 0.01, 0.349],
                             [99.505, 0.028, 0.433],
                             [99.187, 0.296, 0.335],
                             [99.043, 0.224, 0.531],
                             [99.206, 0.166, 0.146],
                             [99.395, 0.188, 0.328],
                             [99.436, 0.199, 0.303],
                             [99.796, 0.009, 0.144],
                             [99.186, 0.397, 0.065],
                             [99.455, 0.079, 0.278],
                             [99.553, 0.084, 0.353],
                             [99.539, 0.017, 0.201],
                             [99.38, 0.082, 0.239],
                             [99.504, 0.009, 0.273],
                             [99.391, 0.261, 0.297],
                             [99.374, 0.015, 0.578],
                             [99.462, 0.179, 0.109],
                             [99.03, 0.213, 0.459],
                             [99.328, 0.131, 0.371],
                             [99.674, 0.055, 0.249],
                             [99.413, 0.137, 0.1],
```

```
                            [99.538, 0.046, 0.151],
                            [99.41, 0.109, 0.08],
                            [99.163, 0.324, 0.343],
                            [99.502, 0.036, 0.412],
                            [99.66, 0.083, 0.069],
                            [99.629, 0.156, 0.069],
                            [99.592, 0.171, 0.008],
                            [99.684, 0.011, 0.106],
                            [99.358, 0.227, 0.137],
                            [99.145, 0.161, 0.403],
                            [99.729, 0.028, 0.123],
                            [99.335, 0.181, 0.351],
                            [99.725, 0.094, 0.14],
                            [99.124, 0.325, 0.015],
                            [99.652, 0.068, 0.029],
                            [99.091, 0.268, 0.565],
                            [99.426, 0.146, 0.256],
                            [99.383, 0.266, 0.039],
                            [99.481, 0.147, 0.327],
                            [99.163, 0.121, 0.71] ] )
        # Initialise the impurity limits & dolar values associated with the different quality grades of Al (aluminium)
        # We require at least a minimum % Al, and no more than max Fe (iron) and Si (Silicon) %'s
        self.no_grades = 11
        self.grade_min_Al=[95.00,99.10,99.10,99.20,99.25,99.35,99.50,99.65,99.75,99.85,99.90]
        self.grade_max_Fe=[ 5.00, 0.81, 0.81, 0.79, 0.76, 0.72, 0.53, 0.50, 0.46, 0.33, 0.30]
        self.grade_max_Si=[ 3.00, 0.40, 0.41, 0.43, 0.39, 0.35, 0.28, 0.28, 0.21, 0.15, 0.15]
        self.grade_value= [10.00,21.25,26.95,36.25,41.53,44.53,48.71,52.44,57.35,68.21,72.56]

    def load_small_problem(self) -> None:
        """Intialise the configuration parameters with default values, and then modify the sizing to give a smaller problem with 10 crucibles"""
        self.load_default_problem()
        self.no_crucibles=10
        self.no_pots=self.no_crucibles * self.pots_per_crucible

    def calc_crucible_value(self, crucible_quality) -> float:
        """Return the $ value of a crucible with the given Al (aluminium), Fe (iron) & Si (silicon) percentages.
           Returns 0 if the aluminium does not satisfy any of the quality grades."""
        tol = 0.00001 # We allow for small errors in 5th decimal point
        for q in reversed(range(self.no_grades)):
            if crucible_quality[Element.Al] >= self.grade_min_Al[q]-tol and \
               crucible_quality[Element.Fe] <= self.grade_max_Fe[q] + tol and \
               crucible_quality[Element.Si] <= self.grade_max_Si[q] + tol:
                return self.grade_value[q]
        return 0.0

    # Calculate the crucible value with a maximum allowed spreaj
    def calc_crucible_value_with_spread(self, crucible_quality, spread: int, max_spread: int) -> float:
        """Return the $ value of a crucible with the given Al (aluminium), Fe (iron) & Si (silicon) percentages.
           Returns 0 if the aluminium does not satisfy any of the quality grades."""
        tol = 0.00001 # We allow for small errors in 5th decimal point
        # spread penalty calcaultion
        spread_penalty = -20*(spread - max_spread) if spread > max_spread else 0
        for q in reversed(range(self.no_grades)):
            if crucible_quality[Element.Al] >= self.grade_min_Al[q]-tol and \
               crucible_quality[Element.Fe] <= self.grade_max_Fe[q] + tol and \
               crucible_quality[Element.Si] <= self.grade_max_Si[q] + tol:
                return self.grade_value[q] + spread_penalty
        return 0.0

    def view_soln(self, x, max_allowed_spread: int=0) -> None:
        """Print solution x with its statistics. Note that our output numbers items from 1, not 0"""
```

```python
        """Print solution x with its statistics. Note that our output numbers items from 1, not 0
        max_spread = 0
        crucible_value_sum = 0
        for c in range (self.no_crucibles):
            spread = max(x[c]) - min(x[c])
            max_spread = max(max_spread, spread)
            crucible_quality = [ (sum( self.pot_quality[x[c][i]][e] for i in range(self.pots_per_crucible) ) / self.pots_per_crucible) for e in Element]
            # max allowed spread functionality added (only calculate with max allowed spread if defined non-zero)
            if max_allowed_spread:
                crucible_value = self.calc_crucible_value_with_spread(crucible_quality, spread, max_allowed_spread)
            else:
                crucible_value = self.calc_crucible_value(crucible_quality)

            crucible_value_sum += crucible_value
            print(f'{c+1:>2} [{x[c][0]+1:>2} {x[c][1]+1:>2} {x[c][2]+1:>2} ] '
                  f'{crucible_quality[Element.Al]:>5.3f} %Al, '
                  f'{crucible_quality[Element.Fe]:>5.3f} %Fe, '
                  f'{crucible_quality[Element.Si]:>5.3f} %Si, '
                  f'${crucible_value:>5.2f}, spread = {spread:>2}' )
        print(f'                                          Sum = ${round(crucible_value_sum,2):>6}, MxSprd = {max_spread:>2}')

    def calc_obj(self, x, max_allowed_spread: int=0):
        """Calculate the total profit for a given solution"""
        crucible_value_sum = 0
        for c in range (self.no_crucibles):
            crucible_quality = [ (sum( self.pot_quality[x[c][i]][e] for i in range(self.pots_per_crucible) ) / self.pots_per_crucible) for e in Element]
            # max allowed spread functionality added (only calculate with max allowed spread if defined non-zero)
            if max_allowed_spread:
                crucible_value = self.calc_crucible_value_with_spread(crucible_quality, np.ptp(x[c]), max_allowed_spread)
            else:
                crucible_value = self.calc_crucible_value ( crucible_quality ) ;
            crucible_value_sum += crucible_value
        return crucible_value_sum

    def trivial_solution(self):
        """Return a solution x=[0,1,2;3,4,5;6,7,8;...;48,49,50] of pots assigned to crucibles"""
        return np.arange(self.no_pots).reshape(self.no_crucibles, self.pots_per_crucible)

    def random_solution(self):
        """Return a random solution of pots assigned to crucibles by shuffling the values in [0,1,2;3,4,5;6,7,8;...;48,49,50] """
        rng = np.random.default_rng()
        x = np.arange(self.no_pots)
        rng.shuffle(x)
        return x.reshape(self.no_crucibles, self.pots_per_crucible)

    def plot_ascent(self, fx, fy, save_name: str, title: str):
        fig = plt.figure()
        plt.plot(fy,'r', label="f(y)")
        plt.plot(fx,'b', label="f(x)")
        plt.xlabel('Function Evaluation Count')
        plt.ylabel('Objective Function Value')
        plt.legend()
        plt.title(title)
        plt.gcf().set_size_inches(11.69, 8.27)
        plt.savefig(f"./report/assets/{save_name}", orientation="landscape")

    ##########
    # TASK 3A #
    ##########
    def next_ascent_to_local_max(self, random_start=True, plotting=False):
        if random_start:
```

```
            x = self.random_solution()
        else:
            x = self.trivial_solution()

        # intermediate values
        last_crucible_values = np.zeros(self.no_crucibles)
        for c in range(self.no_crucibles):
            crucible_quality = [ (sum( self.pot_quality[x[c][i]][e] for i in range(self.pots_per_crucible) ) / self.pots_per_crucible) for e in Element]
            last_crucible_values[c] = self.calc_crucible_value(crucible_quality)

        if plotting:
            fx = []
            fy = []
            fx.append(sum(last_crucible_values))
            fy.append(sum(last_crucible_values))

        # for default case
        last_optimal_indices = (-1, -1, -1, -1)
        while True:
            # loop through neighborhood
            for k in range(self.no_crucibles-1):
                for m in range(self.pots_per_crucible):
                    for l in range(k+1, self.no_crucibles):
                        for n in range(self.pots_per_crucible):

                            # exactly one scan since last optimal value found, can return
                            if (k, m, l, n) == last_optimal_indices:
                                if plotting:
                                    self.plot_ascent(fx, fy, "next_ascent_chart.pdf", "Task 3C")
                                return x

                            # calculate crucible values and delta
                            crucible_k = x[k].copy()
                            crucible_l = x[l].copy()
                            crucible_k[m] = x[l][n]
                            crucible_l[n] = x[k][m]
                            crucible_k_quality = [ (sum( self.pot_quality[crucible_k[i]][e] for i in range(self.pots_per_crucible) ) / self.pots_per_crucible) for e in Element]
                            crucible_k_value = self.calc_crucible_value(crucible_k_quality)
                            crucible_l_quality = [ (sum( self.pot_quality[crucible_l[i]][e] for i in range(self.pots_per_crucible) ) / self.pots_per_crucible) for e in Element]
                            crucible_l_value = self.calc_crucible_value(crucible_l_quality)
                            delta = crucible_k_value + crucible_l_value - last_crucible_values[k] - last_crucible_values[l]

                            if plotting:
                                fy.append(sum(last_crucible_values) + delta)


                            # > 0.001 as don't want to accept new solution if floating point error
                            if delta > 0.001:
                                # update intermediate values, solution, and optimal indices
                                last_optimal_indices = (k, m, l, n)
                                last_crucible_values[k] = crucible_k_value
                                last_crucible_values[l] = crucible_l_value
                                x[k][m] = crucible_k[m]
                                x[l][n] = crucible_l[n]

                            if plotting:
                                fx.append(sum(last_crucible_values))

            # case where starting at local max
            if last_optimal_indices == (-1, -1, -1, -1):
```

```python
                if plotting:
                    self.plot_ascent(fx, fy, "next_ascent_chart.pdf", "Task 3C")
                return x

    ###########
    # TASK 3B #
    ###########
    def steepest_ascent_to_local_max(self, random_start=True, plotting=False):
        if random_start:
            x = self.random_solution()
        else:
            x = self.trivial_solution()

        # intermediate values
        last_crucible_values = np.zeros(self.no_crucibles)
        for c in range(self.no_crucibles):
            crucible_quality = [ (sum( self.pot_quality[x[c][i]][e] for i in range(self.pots_per_crucible) ) / self.pots_per_crucible) for e in Element]
            last_crucible_values[c] = self.calc_crucible_value(crucible_quality)

        if plotting:
            fx = []
            fy = []
            fx.append(sum(last_crucible_values))
            fy.append(sum(last_crucible_values))

        while True:
            optimal_swap = (-1, -1, -1, -1)

            # min starting delta 0.001 for floating point errors
            best_delta = 0.001
            for k in range(self.no_crucibles-1):
                for m in range(self.pots_per_crucible):
                    for l in range(k+1, self.no_crucibles):
                        for n in range(self.pots_per_crucible):

                            # calculate crucible values and delta
                            crucible_k = x[k].copy()
                            crucible_l = x[l].copy()
                            crucible_k[m] = x[l][n]
                            crucible_l[n] = x[k][m]
                            crucible_k_quality = [ (sum( self.pot_quality[crucible_k[i]][e] for i in range(self.pots_per_crucible) ) / self.pots_per_crucible) for e in Element]
                            crucible_k_value = self.calc_crucible_value(crucible_k_quality)
                            crucible_l_quality = [ (sum( self.pot_quality[crucible_l[i]][e] for i in range(self.pots_per_crucible) ) / self.pots_per_crucible) for e in Element]
                            crucible_l_value = self.calc_crucible_value(crucible_l_quality)
                            delta = crucible_k_value + crucible_l_value - last_crucible_values[k] - last_crucible_values[l]

                            if plotting:
                                fy.append(sum(last_crucible_values) + delta)
                                fx.append(sum(last_crucible_values))

                            # if new steepest update best delta and save optimal swap location
                            if delta > best_delta:
                                best_delta = delta
                                optimal_swap = (k, m, l, n)

            # if all neighbors scanned and no better solution found, at local max and finish
            if optimal_swap == (-1, -1, -1, -1):
                if plotting:
                    self.plot_ascent(fx, fy, "steepest_ascent_chart.pdf", "Task 3D")
                return x
```

```python

            # Make swap with steepest neighbor and update intermediate values
            k, m, l, n = optimal_swap
            crucible_k = x[k].copy()
            crucible_l = x[l].copy()
            crucible_k[m] = x[l][n]
            crucible_l[n] = x[k][m]
            crucible_k_quality = [ (sum( self.pot_quality[crucible_k[i]][e] for i in range(self.pots_per_crucible) ) / self.pots_per_crucible) for e in Element]
            crucible_k_value = self.calc_crucible_value(crucible_k_quality)
            crucible_l_quality = [ (sum( self.pot_quality[crucible_l[i]][e] for i in range(self.pots_per_crucible) ) / self.pots_per_crucible) for e in Element]
            crucible_l_value = self.calc_crucible_value(crucible_l_quality)
            last_crucible_values[k] = crucible_k_value
            last_crucible_values[l] = crucible_l_value
            x[k][m] = crucible_k[m]
            x[l][n] = crucible_l[n]


    ###########
    # TASK 3E #
    ###########
    def do_repeated_next_ascents(self, n: int, max_spread: int = 0, plotting=True):
        best_obj_history = []
        obj_history = []
        times = []

        # Iterate through random starts to find history and best solution
        best_obj = 0
        start_time = time.perf_counter()
        for _ in range(n):
            # If max spread specified then do with max spread (for Task 6)
            if max_spread:
                x = self.next_ascent_to_local_max_spread(max_spread)
            else:
                x = self.next_ascent_to_local_max()
            obj = self.calc_obj(x)
            if obj > best_obj:
                best_x = x
                best_obj = obj
            best_obj_history.append(best_obj)
            obj_history.append(obj)
            times.append(time.perf_counter() - start_time)

        # Output and plot best solution
        print(f"repeated next ascents max_spread={max_spread}")
        self.view_soln(best_x)
        if plotting:
            fig = plt.figure()
            plt.scatter(times,best_obj_history,c='b',s=1, label="Best objective value")
            plt.scatter(times,obj_history,c='r',s=5, label="Local max")
            plt.xlabel('Time (s)')
            plt.ylabel('Objective Function Value')
            plt.legend()
            if max_spread:
                plt.title(f"Task 6 Repeated Next Ascents (n={n}, max_spread={max_spread})")
            else:
                plt.title(f"Task 3E Repeated Next Ascents (n={n})")
            plt.gcf().set_size_inches(11.69, 8.27)
            if max_spread:
                plt.savefig(f"./report/assets/repeated_next_ascents_chart__max_spread_{max_spread}.pdf", orientation="landscape")
            else:
                plt.savefig("./report/assets/repeated_next_ascents_chart.pdf", orientation="landscape")
```

```python
                plt.savefig("./report/assets/repeated_next_ascents_chart.pdf", orientation="landscape")

    ###########
    # TASK 3E #
    ###########
    def do_repeated_steepest_ascents(self, n: int):
        best_obj_history = []
        obj_history = []
        times = []

        # Iterate through random starts to find history and best solution
        best_obj = 0
        start_time = time.perf_counter()
        for _ in range(n):
            x = self.steepest_ascent_to_local_max()
            obj = self.calc_obj(x)
            if obj > best_obj:
                best_x = x
                best_obj = obj
            best_obj_history.append(best_obj)
            obj_history.append(obj)
            times.append(time.perf_counter() - start_time)

        # Output and plot best solution
        self.view_soln(best_x)
        fig = plt.figure()
        plt.scatter(times,best_obj_history,c='b',s=1, label="Best objective value")
        plt.scatter(times,obj_history,c='r',s=5, label="Local max")
        plt.xlabel('Time (s)')
        plt.ylabel('Objective Function Value')
        plt.title(f"Task 3E Repeated Steepest Ascents (n={n})")
        plt.legend()
        plt.gcf().set_size_inches(11.69, 8.27)
        plt.savefig("./report/assets/repeated_steepest_ascents_chart.pdf", orientation="landscape")

    ##########
    # TASK 6 #
    ##########
    def next_ascent_to_local_max_spread(self, max_spread: int, random_start=True):
        if random_start:
            x = self.random_solution()
        else:
            x = self.trivial_solution()

        # init intermeidate values
        last_crucible_values = np.zeros(self.no_crucibles)
        for c in range(self.no_crucibles):
            crucible_quality = [ (sum( self.pot_quality[x[c][i]][e] for i in range(self.pots_per_crucible) ) / self.pots_per_crucible) for e in Element]
            last_crucible_values[c] = self.calc_crucible_value_with_spread(crucible_quality, np.ptp(x[c]), max_spread)

        # Loop through neighbors
        last_optimal_indices = (-1, -1, -1, -1)
        while True:
            for k in range(self.no_crucibles-1):
                for m in range(self.pots_per_crucible):
                    for l in range(k+1, self.no_crucibles):
                        for n in range(self.pots_per_crucible):
                            # looped through all neighbors once and no better solution found
                            if (k, m, l, n) == last_optimal_indices:
                                return x
```

```python
                            # calculate delta and other relevant params
                            crucible_k = x[k].copy()
                            crucible_l = x[l].copy()
                            crucible_k[m] = x[l][n]
                            crucible_l[n] = x[k][m]
                            crucible_k_quality = [ (sum( self.pot_quality[crucible_k[i]][e] for i in range(self.pots_per_crucible) ) / self.pots_per_crucible) for e in Element]
                            crucible_k_value = self.calc_crucible_value_with_spread(crucible_k_quality, np.ptp(crucible_k), max_spread)
                            crucible_l_quality = [ (sum( self.pot_quality[crucible_l[i]][e] for i in range(self.pots_per_crucible) ) / self.pots_per_crucible) for e in Element]
                            crucible_l_value = self.calc_crucible_value_with_spread(crucible_l_quality, np.ptp(crucible_l), max_spread)
                            delta = crucible_k_value + crucible_l_value - last_crucible_values[k] - last_crucible_values[l]

                            # better solution so update intermediate values and solution
                            if delta > 0.01:
                                last_optimal_indices = (k, m, l, n)
                                last_crucible_values[k] = crucible_k_value
                                last_crucible_values[l] = crucible_l_value
                                x[k][m] = crucible_k[m]
                                x[l][n] = crucible_l[n]

            # case where already at local max
            if last_optimal_indices == (-1, -1, -1, -1):
                return x

if __name__ == "__main__":
    ls = LocalSearch()
    ls.load_small_problem()
    ls.next_ascent_to_local_max(random_start=False, plotting=True)
    ls.steepest_ascent_to_local_max(random_start=False, plotting=True)
    ls.load_default_problem()
    ls.do_repeated_next_ascents(200)
    ls.do_repeated_steepest_ascents(200)
    ls.do_repeated_next_ascents(200, max_spread=6, plotting=False)
    ls.do_repeated_next_ascents(200, max_spread=8, plotting=False)
    ls.do_repeated_next_ascents(200, max_spread=11, plotting=False)
```