Mastermind with n colors

0.1 Problem setting

We consider an array of length N, each cell contains a number in $\{0, 1, \ldots, \text{N-1}\}$.

We can compare another array with this array, the integer returned is the number of equal cells.

The purpose of the algorithms we study is to find this array using a minimum number of comparison with it.

0.2 Exhaustive-Search

0.2.1 Algorithm

The first algorithm is an exhaustive search

while *F(S) != N* **do**
  $S \leftarrow$ next array;
  Evaluate $F(S)$;
end

Algorithm 1: Exhaustive Search

0.2.2 Result

size	per 0.02	per 0.25	per 0.5	per 0.75	per 0.98	mean	std dev	count
2	1.0	1	2	3	4.0	2.378	1.134	45
4	11.52	76	164	199	241.24	141.6	68.785	45
8	1255927.0	6163844	7771397	13098828	14896442.88	8632855.889	4858003.181	9

The complexity is roughly what we expect : $\Theta(N^N)$.

0.3 Erdos-Renyi

0.3.1 Algorithm

begin
  Pick a Set S of random array in $\{0, ..., N-1\}^N$ to Evaluate.
  foreach *A in S* **do**
    compute F(A)
  end
  Create an empty set of possible solution S′ **foreach** *Possible Array A′* **do**
    foreach *A in S* **do**
      if *distance(A, A′) != F(A)* **then**
        A′ can't be solution
        Continue to next iteration step
      end
    end
    Add A′ to S′
  end
end

Algorithm 2: Erdos Renyi Algorithm

With a big enough value of $|S|$ (see Erdos-Renyi study), we have $|S'| = 1$ with high probability.

0.3.2 Result

size	per 0.02	per 0.25	per 0.5	per 0.75	per 0.98	mean	std dev	count
2	1.0	1	2.0	3	5	2.3	1.317	30
4	3.58	5	6.0	7	9	6.033	1.45	30
8	17.0	17	17.0	17	17	17.0		1

We try to adapt the Erdos-Renyi algorithm to N colors. The algorithm yields good result in term of number of call to the comparison function but the overall time needed is in $O(N^N)$.

0.4 Random Local Search

0.4.1 Algorithm

$S \leftarrow$ random array **while** *F(S) != N* **do**
  $i \leftarrow$ randint(0, N-1)
  $S' \leftarrow S$
  $S'[i] \leftarrow step()$
  if *F(S′) > F(S)* **then**
    $S \leftarrow S'$
  end
end

Algorithm 3: Random Local Search

0.4.2 Result

size	per 0.02	per 0.25	per 0.5	per 0.75	per 0.98	mean	std dev	count
2	1.0	2.0	3	4.0	5.0	2.8	1.373	15
4	5.68	10.5	13	24.5	41.72	18.4	11.667	15
8	67.12	109.5	159	215.0	285.8	161.733	70.673	15
16	501.0	714.5	848	981.5	1524.88	883.0	288.114	15
32	2078.4	4093.0	4276	4723.5	6293.48	4250.667	1129.859	15
64	13247.56	14633.5	20937	24208.0	27993.96	19967.133	5379.733	15
128	66483.76	75320.5	83684	87236.5	117226.24	84380.933	14373.753	15
256	313078.72	358965.0	382312	417418.0	494008.96	394791.8	56323.934	15

The experimental results seems to show that the overall complexity is in $N \cdot N \cdot log(N)$

1 Evolutionary Algorithm

We now study evolutionary algorithm. The step function used for all our algorithm consist of picking a random new color for a cell.

1.1 (1+1) Evolution Algorithm

1.1.1 Algorithm

$S \leftarrow$ random array
while *F(S) !=* N **do**
  $S' \leftarrow S$
  for $i = 0;\ i < N;\ i++$ **do**
    if *random() <= mutation_rate* **then**
      $S'[i] \leftarrow step()$
    end
  end
  if *F(S') > F(S)* **then**
    $S \leftarrow S'$
  end
end

Algorithm 4: (1+1)EA

1.1.2 Result

size	mu	lambda	mutation rate	per 0.02	per 0.25	per 0.5	per 0.75	per 0.98	mean	std dev	count
2	1	1	0.5	1.0	1.0	3	3.5	15.32	4.067	4.667	15
4	1	1	0.25	9.68	19.5	34	40.5	74.76	33.333	19.059	15
8	1	1	0.125	122.24	186.5	234	318.5	571.32	288.333	155.646	15
16	1	1	0.062	1307.76	1663.0	2269	3031.0	4954.72	2499.933	1102.725	15
32	1	1	0.031	5534.24	7497.0	8427	11231.5	16616.04	9440.4	3299.962	15
64	1	1	0.016	28051.6	37844.5	41506	46798.5	70748.48	44194.533	12229.526	15
128	1	1	0.008	143334.24	160700.5	211083	225974.0	253009.92	197100.333	39450.459	15
256	1	1	0.004	867953.0	867953.0	867953	867953.0	867953.0	867953.0		1

1.2 $(\mu + \lambda)$Genetic Algorithm

$S \leftarrow$ Set of μ random array
while *max(F(S)) != N* **do**
  $S' \leftarrow$ empty Set
  for $i = 0;\ i < \lambda;\ i++$ **do**
    $A \leftarrow$ majority_vote(S, k)
    for $j = 0;\ j < N;\ j++$ **do**
      if *random() <= mutation_rate* **then**
        $A[i] \leftarrow step()$
      end
      Add A to S'
    end
    $S \leftarrow$ Elitist_Selection(S, S')
  end
end

Algorithm 5: $(\mu + \lambda)$GA

1.2.1 Result

size	mu	lambda	mutation rate	per 0.02	per 0.25	per 0.5	per 0.75	per 0.98	mean	std dev	count
2	2	1	0.5	2.0	2.5	3	4.5	15.48	4.667	4.135	15
2	10	6	0.5	10.0	13.0	16	16.0	16.0	14.4	2.746	15
4	2	1	0.25	18.56	23.0	27	45.5	72.56	35.267	17.392	15
4	10	6	0.25	17.68	40.0	46	61.0	78.64	48.4	17.683	15
8	2	1	0.125	122.56	134.0	177	326.0	481.64	242.933	133.195	15
8	10	6	0.125	139.36	172.0	184	241.0	403.12	217.6	79.074	15
16	2	1	0.062	839.8	963.0	1342	2351.5	2625.92	1552.2	729.231	15
16	10	6	0.062	526.72	793.0	1018	1216.0	1624.48	1033.6	334.557	15
32	2	1	0.031	3719.64	4577.5	5345	7144.0	13422.76	6537.533	3094.985	15
32	10	6	0.031	3300.64	4024.0	5026	6439.0	9895.12	5548.4	2015.521	15
64	2	1	0.016	20285.48	28036.5	30694	31764.0	42031.84	30862.267	6039.868	15
64	10	6	0.016	20412.64	22060.0	25858	30442.0	59236.72	30342.4	12771.794	15
128	2	1	0.008	168539.0	168539.0	168539	168539.0	168539.0	168539.0		1
128	10	6	0.008	125638.0	125638.0	125638	125638.0	125638.0	125638.0		1

Having a bigger value for μ and λ doesn't seems to yield better result. Maybe because the mutation rate was set to be equals to $\frac{1}{lambda}$.

Value of μ and λ are not easy to choose. In some case, I believe having a smaller value for μ speed up the algorithm but may makes good assignment of a cell being forgotten (for example, 1 good cell change to be wrong, but 2 wrong changes to be good). Having a larger value for μ means older value are not immediately deleted, so in the same case, the information of the good cell which was changed may be retrieved, but it means in some case only not so good will be used in the majority vote of A and the generated offspring may not be the best because because of that.

I believe the $1 + (\lambda, \lambda)GA$ we present now get the best of both worlds.

1.3 $1 + (\lambda, \lambda)GA$

Lastly we tried to adapt the $1 + (\lambda, \lambda)GA$ of the "Optimal Parameter Choices Through Self-Adjustment: Applying the 1/5-th Rule in Discrete Settings" to our case.

1.3.1 Algorithm

$S \leftarrow$ random array
while *F(S) !=* N **do**
  Mutation phase
  begin
    for $i = 0;\ i < \lambda;\ i++$ **do**
      $X_i \leftarrow S$ **for** $j = 0;\ j < N;\ j++$ **do**
        if *random()* <= *mutation_rate* **then**
          $X_i[j] \leftarrow step()$
        end
      end
      Compute(F(X_i))
    end
    $X \leftarrow X_i$ such that $f(X)$ is maximized
  end
  Crossover phase
  begin
    for $i = 0;\ i < \lambda;\ i++$ **do**
      for $j = 0;\ j < N;\ j++$ **do**
        $Y_i[j] \leftarrow cross_c(S, X)$
      end
      Compute(F(Y_i))
    end
    $Y \leftarrow Y_i$ such that $f(Y_i)$ is maximized
  end
  Optional Auto Adjustment Rate begin
    if *F(Y) > F(S)* **then**
      $\lambda \leftarrow \max(\lambda / F, 1)$
    end
    else
      $\lambda \leftarrow \min(\lambda F^{1/4}, N)$
    end
  end
  if *F(Y) > F(S)* **then**
    S $\leftarrow Y$
  end
end

Algorithm 6: $1 + (\lambda, \lambda)$GA

Where F is the update strength, c is the crossover rate and is equals to $\frac{1}{\lambda}$, and the *mutation_rate* is equals to $\frac{\lambda}{N}$.

The function $cross_c(x, y)$ output an array z and $z[i] = x[i]$ with probability c and $z[i] = y[i]$ otherwise.

In the adaptive version, c and *mutate_rate* change with the value of λ. In the case of the non-adaptive version, they remains the same (the initial value).

1.3.2 Result

Non-adaptive version

size	lambda	mutation rate	crossover probability	per 0.02	per 0.25	per 0.5	per 0.75	per 0.98	mean	std dev	count
2	4	1.0	0.25	1.0	5	9	17	36.52	12.2	10.818	15
2	8	1.0	0.125	1.0	1	17	41	95.08	27.667	30.035	15
2	12	1.0	0.083	1.0	25	25	61	97.0	42.6	33.288	15
4	4	1.0	0.25	17.0	25	41	49	99.56	43.133	24.465	15
4	8	1.0	0.125	33.0	49	65	81	124.52	69.267	29.913	15
4	12	1.0	0.083	49.0	61	73	73	155.56	76.2	31.248	15
8	4	0.5	0.25	139.24	193	273	329	638.12	292.2	148.279	15
8	8	1.0	0.125	117.48	153	209	249	323.56	209.0	64.569	15
8	12	1.0	0.083	127.72	217	241	337	419.56	266.6	90.013	15
16	4	0.25	0.25	781.8	1429	1881	2201	3464.68	1943.933	794.782	15
16	8	0.5	0.125	1173.48	1393	1793	2433	4527.72	2122.6	1029.748	15
16	12	0.75	0.083	933.16	1105	1705	2233	3712.36	1852.2	861.323	15
32	4	0.125	0.25	5438.12	7933	10001	13081	14486.12	10324.733	2968.613	15
32	8	0.25	0.125	6578.92	7745	9057	9761	13406.44	9096.467	1974.149	15
32	12	0.375	0.083	5886.76	7537	9049	12325	17554.6	10114.6	3637.72	15
64	4	0.062	0.25	32453.8	39401	41993	56233	63748.52	46511.933	10436.636	15
64	8	0.125	0.125	35830.12	39417	40721	53049	60246.76	45486.867	8754.413	15
64	12	0.188	0.083	27965.8	40561	46993	52933	89598.76	49061.8	17192.228	15
128	4	0.031	0.25	162769.0	185665	201921	241385	362813.48	223105.0	60037.529	15
128	8	0.062	0.125	145112.68	190601	211649	248521	304951.4	217258.6	48584.418	15
128	12	0.094	0.083	147265.96	166501	221041	251521	313345.96	215831.4	53732.739	15
256	4	0.016	0.25	682281.0	682281	682281	682281	682281.0	682281.0		1
256	8	0.031	0.125	1205761.0	1205761	1205761	1205761	1205761.0	1205761.0		1
256	12	0.047	0.083	897889.0	897889	897889	897889	897889.0	897889.0		1

Adaptive version

size	update strength	per 0.02	per 0.25	per 0.5	per 0.75	per 0.98	mean	std dev	count
2	1.1	1.0	1	5	13	25.0	7.667	8.902	15
4	1.1	11.24	25	41	69	124.52	50.6	34.502	15
8	1.1	101.48	161	195	249	348.52	210.2	71.896	15
16	1.1	786.52	905	997	1257	1390.6	1066.067	215.396	15
32	1.1	4150.68	4546	4927	5245	6254.12	4921.933	629.403	15
64	1.1	17671.64	19474	21111	22398	22991.72	20761.4	1822.801	15
128	1.1	89014.68	90249	91869	93332	98975.72	92410.867	2985.324	15
256	1.1	373531.0	373531	373531	373531	373531.0	373531.0		1

In the case of the non-adaptive version, having a large λ doesn't seems to yield significant better results than having a smaller one (keep in mind the number of experiment is equals to 15).

The adaptive case have better results than the other case. I believe the reason is that at all stages of the algorithm, the value of the λ is changed to have step with better performances: a small lambda at the early stage of the algorithm because when most of the cells aren't equals, changing only a few cells means one of them is probably false and hence we have good chance of finding the real value of a cell.

In the late stage of the algorithm, I believe the value of the lambda increase, hence we have more mutation and more offspring. Among these offspring, the best offspring X is probably a poor choice in itself (because a lot of correct values of S where mutated) and the cross function will have a small value for c so we are almost ensured to keep the good values of the initial S and only change the new one found by X.

2 Conclusion

To conclude for all the evolutionary algorithm the complexity seems to remains in $O(N \cdot N \cdot log(N))$, only the adaptive version have experimental results close to the random local search.