

## Graph Coloring Algorithms

**Sampling Coloring:** A simple heuristic method that tries random colorings and keeps the best one found. It is fast but may not find the optimal solution.

**Brute Force Coloring:** Guarantees finding the optimal solution but is computationally expensive since it explores all possible colorings.

**Deterministic Hill Climbing:** Improves over random search by always selecting the best neighboring solution but can get stuck in local optima.

**Stochastic Hill Climbing:** Adds randomness to avoid local optima, making it more flexible than deterministic hill climbing.

**Tabu Search:** A more sophisticated approach that avoids cycling and can explore a broader solution space, but it is more complex and computationally intensive.

---

### Algorithm 1 Sampling Coloring

---

**Require:** Graph  $G$ , Number of Samples  $n$

**Ensure:** Best Coloring, Best Loss, Attempts

```
1:  $initial\_colors \leftarrow \frac{|nodes(G)|}{2}$ 
2:  $best\_coloring \leftarrow generate\_random\_coloring(G, initial\_colors)$ 
3:  $best\_loss \leftarrow calculate\_loss(G, best\_coloring)$ 
4:  $attempts \leftarrow 0$ 
5: for  $i = 1$  to  $n$  do
6:    $attempts \leftarrow attempts + 1$ 
7:    $coloring \leftarrow generate\_random\_coloring(G, initial\_colors)$ 
8:    $loss \leftarrow calculate\_loss(G, coloring)$ 
9:   if  $loss < best\_loss$  then
10:     $best\_loss \leftarrow loss$ 
11:     $best\_coloring \leftarrow coloring$ 
12:   end if
13: end for
14: return  $best\_coloring, best\_loss, attempts$ 
```

---

---

**Algorithm 2** Brute Force Coloring

---

**Require:** Graph  $G$ , Maximum Colors  $max\_colors$ **Ensure:** Best Coloring, Best Loss, Attempts

```
1:  $nodes \leftarrow list(nodes(G))$ 
2:  $best\_coloring \leftarrow None, best\_loss \leftarrow \infty$ 
3:  $attempts \leftarrow 0, min\_colors \leftarrow 1$ 
4: for  $num\_colors = min\_colors$  to  $max\_colors$  do
5:   for each assignment of colors to nodes do
6:      $attempts \leftarrow attempts + 1$ 
7:      $coloring \leftarrow map\_nodes\_to\_colors(nodes, num\_colors)$ 
8:      $loss \leftarrow calculate\_loss(G, coloring)$ 
9:     if  $loss < best\_loss$  then
10:       $best\_loss \leftarrow loss$ 
11:       $best\_coloring \leftarrow coloring$ 
12:      if  $loss == 0$  then
13:        return  $best\_coloring, best\_loss, attempts$ 
14:      end if
15:    end if
16:  end for
17: end for
18: return  $best\_coloring, best\_loss, attempts$ 
```

---

---

**Algorithm 3** Deterministic Hill Climbing

---

**Require:** Graph  $G$

**Ensure:** Best Coloring, Best Loss, Attempts

```
1:  $num\_colors \leftarrow \frac{|nodes(G)|}{2}$ 
2:  $current \leftarrow generate\_random\_coloring(G, num\_colors)$ 
3:  $current\_loss \leftarrow calculate\_loss(G, current)$ 
4:  $attempts \leftarrow 1$ 
5: while true do
6:    $best\_neighbor \leftarrow current$ 
7:    $best\_neighbor\_loss \leftarrow current\_loss$ 
8:   for each node in  $G$  do
9:     for each possible color do
10:       $neighbor \leftarrow change\_color(current, node, color)$ 
11:       $neighbor\_loss \leftarrow calculate\_loss(G, neighbor)$ 
12:       $attempts \leftarrow attempts + 1$ 
13:      if  $neighbor\_loss < best\_neighbor\_loss$  then
14:         $best\_neighbor \leftarrow neighbor$ 
15:         $best\_neighbor\_loss \leftarrow neighbor\_loss$ 
16:      end if
17:    end for
18:  end for
19:  if  $best\_neighbor\_loss \geq current\_loss$  then
20:    return  $current, current\_loss, attempts$ 
21:  end if
22:   $current \leftarrow best\_neighbor$ 
23:   $current\_loss \leftarrow best\_neighbor\_loss$ 
24: end while
```

---

---

**Algorithm 4** Stochastic Hill Climbing

---

**Require:** Graph  $G$ , Max Attempts  $max\_attempts$

**Ensure:** Best Coloring, Best Loss, Attempts

```
1:  $num\_colors \leftarrow \frac{|nodes(G)|}{2}$ 
2:  $current \leftarrow generate\_random\_coloring(G, num\_colors)$ 
3:  $current\_loss \leftarrow calculate\_loss(G, current)$ 
4:  $attempts \leftarrow 1$ 
5: while  $attempts < max\_attempts$  do
6:    $neighbor \leftarrow get\_neighbor(G, current, num\_colors)$ 
7:    $neighbor\_loss \leftarrow calculate\_loss(G, neighbor)$ 
8:    $attempts \leftarrow attempts + 1$ 
9:   if  $neighbor\_loss < current\_loss$  then
10:     $current \leftarrow neighbor$ 
11:     $current\_loss \leftarrow neighbor\_loss$ 
12:    if  $current\_loss == 0$  then
13:      end if
14:    end if
15: end while
16: return  $current, current\_loss, attempts$ 
```

---

---

**Algorithm 5** Tabu Search for Graph Coloring

---

**Require:** Graph  $G$ , Tabu List Size  $tabu\_size$ , Max Iterations  $max\_iterations$ **Ensure:** Best Coloring, Best Loss, Attempts

```
1:  $num\_colors \leftarrow \frac{|nodes(G)|}{2}$ 
2:  $current \leftarrow generate\_random\_coloring(G, num\_colors)$ 
3:  $current\_loss \leftarrow calculate\_loss(G, current)$ 
4:  $best\_solution \leftarrow current$ 
5:  $best\_loss \leftarrow current\_loss$ 
6:  $tabu\_list \leftarrow []$ 
7:  $attempts \leftarrow 1$ 
8: while  $attempts < max\_iterations$  and  $current\_loss > 0$  do
9:    $best\_neighbor \leftarrow None$ 
10:   $best\_neighbor\_loss \leftarrow \infty$ 
11:   $best\_move \leftarrow None$ 
12:  for each node in  $nodes(G)$  do
13:    for each color in  $range(num\_colors)$  do
14:      if  $color \neq current[node]$  then
15:         $move \leftarrow (node, current[node], color)$ 
16:        if  $move \notin tabu\_list$  then
17:           $neighbor \leftarrow current.copy()$ 
18:           $neighbor[node] \leftarrow color$ 
19:           $neighbor\_loss \leftarrow calculate\_loss(G, neighbor)$ 
20:           $attempts \leftarrow attempts + 1$ 
21:          if  $neighbor\_loss < best\_neighbor\_loss$  then
22:             $best\_neighbor \leftarrow neighbor$ 
23:             $best\_neighbor\_loss \leftarrow neighbor\_loss$ 
24:             $best\_move \leftarrow move$ 
25:          end if
26:        end if
27:      end if
28:    end for
29:  end for
30:  if  $best\_neighbor \neq None$  then
31:     $current \leftarrow best\_neighbor$ 
32:     $current\_loss \leftarrow best\_neighbor\_loss$ 
33:    if  $current\_loss < best\_loss$  then
34:       $best\_solution \leftarrow current$ 
35:       $best\_loss \leftarrow current\_loss$ 
36:    end if
37:     $tabu\_list \leftarrow tabu\_list + [best\_move]$ 
38:    if  $length\ of\ tabu\_list \geq tabu\_size$  then
39:       $tabu\_list.pop(0)$ 
40:    end if
41:  else
42:    BREAK
43:  end if
44: end while
45: return  $best\_solution, best\_loss, attempts$ 
```

---

---

**Algorithm 6** Simulated Annealing for Graph Coloring

---

**Require:** Graph  $G$ , Initial Temperature  $initial\_temp$ , Minimum Temperature  $min\_temp$ , Max Iterations  $max\_iterations$ , Cooling Schedule  $schedule$

**Ensure:** Best Coloring, Best Loss, Attempts

```
1:  $num\_colors \leftarrow \frac{|nodes(G)|}{2}$ 
2:  $current \leftarrow generate\_random\_coloring(G, num\_colors)$ 
3:  $current\_loss \leftarrow calculate\_loss(G, current)$ 
4:  $best\_solution \leftarrow current$ 
5:  $best\_loss \leftarrow current\_loss$ 
6:  $temperature \leftarrow initial\_temp$ 
7:  $attempts \leftarrow 1$ 
8: while  $temperature > min\_temp$  and  $attempts < max\_iterations$  and  $current\_loss > 0$  do
9:    $neighbor \leftarrow get\_gaussian\_neighbor(G, current, num\_colors)$ 
10:   $neighbor\_loss \leftarrow calculate\_loss(G, neighbor)$ 
11:   $attempts \leftarrow attempts + 1$ 
12:   $delta \leftarrow neighbor\_loss - current\_loss$ 
13:  if  $delta < 0$  or  $random() \leq \exp(-delta / temperature)$  then
14:     $current \leftarrow neighbor$ 
15:     $current\_loss \leftarrow neighbor\_loss$ 
16:    if  $current\_loss < best\_loss$  then
17:       $best\_solution \leftarrow current$ 
18:       $best\_loss \leftarrow current\_loss$ 
19:    end if
20:  end if
21:   $temperature \leftarrow get\_temperature(initial\_temp, attempts, max\_iterations, schedule)$ 
22: end while
23: return  $best\_solution, best\_loss, attempts$ 
```

---

---

**Algorithm 7** Genetic Algorithm for Graph Coloring

---

**Require:** Graph  $G$ , Population Size  $population\_size$ , Elite Size  $elite\_size$ , Max Generations  $max\_generations$ , Crossover Type  $crossover\_type$ , Mutation Type  $mutation\_type$ , Termination Type  $termination\_type$

**Ensure:** Best Coloring, Best Loss, Attempts

```
1:  $num\_colors \leftarrow \frac{|nodes(G)|}{2}$ 
2:  $population \leftarrow generate\_initial\_population(G, population\_size, num\_colors)$ 
3:  $attempts \leftarrow population\_size$ 
4:  $best\_solution \leftarrow \max(population, key fitness)$ 
5:  $generation \leftarrow 0$ 
6: while termination condition not met do
7:   Sort  $population$  by fitness in descending order
8:   if  $best\_solution.fitness = 0$  then
9:     return  $best\_solution.coloring, 0, attempts$ 
10:  end if
11:  if termination type is generations and  $generation \geq max\_generations$  then
12:    return  $best\_solution.coloring, -best\_solution.fitness, attempts$ 
13:  end if
14:   $new\_population \leftarrow population[: elite\_size]$ 
15:  while size of  $new\_population < population\_size$  do
16:     $parent1, parent2 \leftarrow selectparentsfromtophalfofpopulation$ 
17:    if  $crossover\_type = CrossoverType.UNIFORM$  then
18:       $child1, child2 \leftarrow uniform\_crossover(parent1, parent2)$ 
19:    else
20:       $child1, child2 \leftarrow single\_point\_crossover(parent1, parent2)$ 
21:    end if
22:    if  $mutation\_type = MutationType.RANDOM$  then
23:       $child1 \leftarrow random\_mutation(child1, num\_colors)$ 
24:       $child2 \leftarrow random\_mutation(child2, num\_colors)$ 
25:    else
26:       $child1 \leftarrow swap\_mutation(child1)$ 
27:       $child2 \leftarrow swap\_mutation(child2)$ 
28:    end if
29:     $fitness1 \leftarrow -calculate\_loss(G, child1)$ 
30:     $fitness2 \leftarrow -calculate\_loss(G, child2)$ 
31:     $attempts \leftarrow attempts + 2$ 
32:    Add  $child1, child2$  to  $new\_population$ 
33:    if  $fitness1 = 0$  or  $fitness2 = 0$  then
34:      return best solution with fitness 0,  $best\_solution.coloring, 0, attempts$ 
35:    end if
36:  end while
37:  Update population to  $new\_population$ 
38:   $best\_solution \leftarrow \max(population, key fitness)$ 
39:   $generation \leftarrow generation + 1$ 
40: end while
41: return  $best\_solution.coloring, -best\_solution.fitness, attempts$ 
```

---

---

**Algorithm 8** Parallel Genetic Algorithm for Graph Coloring

---

**Require:** Graph  $G$ , Population Size  $population\_size$ , Elite Size  $elite\_size$ , Max Generations  $max\_generations$ , Crossover Type  $crossover\_type$ , Mutation Type  $mutation\_type$ , Termination Type  $termination\_type$ , Number of Processes  $num\_processes$

**Ensure:** Best Coloring, Best Loss, Attempts

```
1:  $num\_colors \leftarrow \frac{|nodes(G)|}{2}$ 
2:  $attempts \leftarrow 0$ 
3: if  $num\_processes$  is None then
4:    $num\_processes \leftarrow cpu\_count()$ 
5: end if
6: INITIALIZE multiprocessing pool with  $num\_processes$ 
7:  $coloring \leftarrow [generate\_random\_coloring(G, num\_colors) \text{ for } i = 1 \text{ to } population\_size]$ 
8:  $fitnesses \leftarrow evaluate\_population\_parallel(G, coloring, pool)$ 
9:  $attempts \leftarrow attempts + population\_size$ 
10:  $population \leftarrow [Individual(c, f) \text{ for } c, f \text{ in } zip(coloring, fitnesses)]$ 
11:  $best\_solution \leftarrow \max(population, key = \lambda x : x.fitness)$ 
12:  $generation \leftarrow 0$ 
13: while termination condition not met do
14:   SORT  $population$  by fitness in descending order
15:   if  $best\_solution.fitness = 0$  then
16:
17:     return  $best\_solution.coloring, 0, attempts$ 
18:   end if
19:   if  $termination\_type = TerminationType.GENERATIONS$  and  $generation \geq max\_generations$  then
20:
21:     return  $best\_solution.coloring, -best\_solution.fitness, attempts$ 
22:   end if
23:    $new\_population \leftarrow population[: elite\_size]$ 
24:    $offspring\_colorings \leftarrow []$ 
25:   while  $|new\_population| + |offspring\_colorings|/2 < population\_size$  do
26:      $parent1, parent2 \leftarrow random.sample(population[: population\_size/2], 2)$ 
27:     if  $crossover\_type = CrossoverType.UNIFORM$  then
28:        $child1, child2 \leftarrow uniform\_crossover(parent1, parent2)$ 
29:     else
30:        $child1, child2 \leftarrow single\_point\_crossover(parent1, parent2)$ 
31:     end if
32:     if  $mutation\_type = MutationType.RANDOM$  then
33:        $child1 \leftarrow random\_mutation(child1, num\_colors)$ 
34:        $child2 \leftarrow random\_mutation(child2, num\_colors)$ 
35:     else
36:        $child1 \leftarrow swap\_mutation(child1)$ 
37:        $child2 \leftarrow swap\_mutation(child2)$ 
38:     end if
39:      $offspring\_colorings.append(child1, child2)$ 
40:   end while
41:    $offspring\_fitnesses \leftarrow evaluate\_population\_parallel(G, offspring\_colorings, pool)$ 
42:    $attempts \leftarrow attempts + |offspring\_fitnesses|$ 
43:   for  $i = 0$  to  $|offspring\_colorings| - 1$  step 2 do
44:      $child1\_coloring \leftarrow offspring\_colorings[i]$ 
45:      $child2\_coloring \leftarrow offspring\_colorings[i + 1]$ 
46:      $child1\_fitness \leftarrow offspring\_fitnesses[i]$ 
47:      $child2\_fitness \leftarrow offspring\_fitnesses[i + 1]$ 
48:     if  $child1\_fitness = 0$  or  $child2\_fitness = 0$  then
49:        $best\_coloring \leftarrow child1\_coloring$  if  $child1\_fitness = 0$  else
50:          $child2\_coloring$ 
```



---

**Algorithm 9** Island Model Genetic Algorithm for Graph Coloring

---

**Require:** Graph  $G$ , Number of Islands  $num\_islands$ , Migration Rate  $migration\_rate$ , Migration Interval  $migration\_interval$ , Population Size  $population\_size$ , Elite Size  $elite\_size$ , Max Generations  $max\_generations$ , Crossover Type  $crossover\_type$ , Mutation Type  $mutation\_type$ , Termination Type  $termination\_type$ , Number of Processes  $num\_processes$

**Ensure:** Best Coloring, Best Loss, Attempts

```
1:  $num\_colors \leftarrow \frac{|nodes(G)|}{2}$ 
2:  $attempts \leftarrow 0$ 
3:  $island\_size \leftarrow \frac{population\_size}{num\_islands}$ 
4: if  $num\_processes$  is None then
5:    $num\_processes \leftarrow cpu\_count()$ 
6: end if
7: INITIALIZE multiprocessing pool with  $num\_processes$ 
8: INITIALIZE islands with  $num\_islands$  each having  $island\_size$  individuals
9: for each island  $i$  do
10:    $coloring \leftarrow [generate\_random\_coloring(G, num\_colors) \text{ for } i = 1 \text{ to } island\_size]$ 
11:    $fitnesses \leftarrow evaluate\_population\_parallel(G, coloring, pool)$ 
12:    $attempts \leftarrow attempts + island\_size$ 
13:    $population \leftarrow [Individual(c, f) \text{ for } c, f \text{ in } zip(coloring, fitnesses)]$ 
14:    $islands.append(population)$ 
15: end for
16:  $best\_solution \leftarrow \max(\max(island, key = \lambda x : x.fitness) \text{ for } island \text{ in } islands)$ 
17:  $generation \leftarrow 0$ 
18: while termination condition not met do
19:   for each island  $i$  do
20:     SORT  $islands[i]$  by fitness in descending order
21:      $current\_best \leftarrow \max(islands[i], key = \lambda x : x.fitness)$ 
22:     if  $current\_best.fitness > best\_solution.fitness$  then
23:        $best\_solution \leftarrow current\_best$ 
24:     end if
25:     if  $best\_solution.fitness = 0$  then
26:       return  $best\_solution.coloring, 0, attempts$ 
27:     end if
28:   if  $termination\_type = TerminationType.GENERATIONS$  and  $generation \geq max\_generations$  then
29:     return  $best\_solution.coloring, -best\_solution.fitness, attempts$ 
30:   end if
31:    $new\_population \leftarrow islands[i][: elite\_size]$ 
32:    $offspring\_colorings \leftarrow []$ 
33:   while  $|new\_population| + |offspring\_colorings|/2 < island\_size$  do
34:      $parent1, parent2 \leftarrow random.sample(islands[i][: island\_size/2], 2)$ 
35:     if  $crossover\_type = CrossoverType.UNIFORM$  then
36:        $child1, child2 \leftarrow uniform\_crossover(parent1, parent2)$ 
37:     else
38:        $child1, child2 \leftarrow single\_point\_crossover(parent1, parent2)$ 
39:     end if
40:     if  $mutation\_type = MutationType.RANDOM$  then
41:        $child1 \leftarrow random\_mutation(child1, num\_colors)$ 
42:        $child2 \leftarrow random\_mutation(child2, num\_colors)$ 
43:     else
44:        $child1 \leftarrow swap\_mutation(child1)$ 
45:        $child2 \leftarrow swap\_mutation(child2)$ 
46:     end if
47:      $offspring\_colorings.append(child1, child2)$ 
```