# 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

$initial\_colors \leftarrow \frac{|nodes(G)|}{2}$
$best\_coloring \leftarrow generate\_random\_coloring(G, initial\_colors)$
$best\_loss \leftarrow calculate\_loss(G, best\_coloring)$
$attempts \leftarrow 0$
**for** $i = 1$ to $n$ **do**
  $attempts \leftarrow attempts + 1$
  $coloring \leftarrow generate\_random\_coloring(G, initial\_colors)$
  $loss \leftarrow calculate\_loss(G, coloring)$
  **if** $loss < best\_loss$ **then**
    $best\_loss \leftarrow loss$
    $best\_coloring \leftarrow coloring$
  **end if**
**end for**
**return** $best\_coloring, best\_loss, attempts$

---

**Algorithm 2** Brute Force Coloring

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

```
nodes ← list(nodes(G))
best_coloring ← None, best_loss ← ∞
attempts ← 0, min_colors ← 1
for num_colors = min_colors to max_colors do
    for each assignment of colors to nodes do
        attempts ← attempts + 1
        coloring ← map_nodes_to_colors(nodes, num_colors)
        loss ← calculate_loss(G, coloring)
        if loss < best_loss then
            best_loss ← loss
            best_coloring ← coloring
            if loss == 0 then
                return best_coloring, best_loss, attempts
            end if
        end if
    end for
end for
return best_coloring, best_loss, attempts
```

**Algorithm 3** Deterministic Hill Climbing

**Require:** Graph $G$

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

```
num_colors ← |nodes(G)| / 2
current ← generate_random_coloring(G, num_colors)
current_loss ← calculate_loss(G, current)
attempts ← 1
while true do
    best_neighbor ← current
    best_neighbor_loss ← current_loss
    for each node in G do
        for each possible color do
            neighbor ← change_color(current, node, color)
            neighbor_loss ← calculate_loss(G, neighbor)
            attempts ← attempts + 1
            if neighbor_loss < best_neighbor_loss then
                best_neighbor ← neighbor
                best_neighbor_loss ← neighbor_loss
            end if
        end for
    end for
    if best_neighbor_loss ≥ current_loss then
        return current, current_loss, attempts
    end if
    current ← best_neighbor
    current_loss ← best_neighbor_loss
end while
```

**Algorithm 4** Stochastic Hill Climbing

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

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

$num\_colors \leftarrow \frac{|nodes(G)|}{2}$
$current \leftarrow generate\_random\_coloring(G, num\_colors)$
$current\_loss \leftarrow calculate\_loss(G, current)$
$attempts \leftarrow 1$
**while** $attempts < max\_attempts$ **do**
  $neighbor \leftarrow get\_neighbor(G, current, num\_colors)$
  $neighbor\_loss \leftarrow calculate\_loss(G, neighbor)$
  $attempts \leftarrow attempts + 1$
  **if** $neighbor\_loss < current\_loss$ **then**
    $current \leftarrow neighbor$
    $current\_loss \leftarrow neighbor\_loss$
    **if** $current\_loss == 0$ **then**
    **end if**
  **end if**
**end while**
**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

```
num_colors ← |nodes(G)| / 2
current ← generate_random_coloring(G, num_colors)
current_loss ← calculate_loss(G, current)
best_solution ← current
best_loss ← current_loss
tabu_list ← []
attempts ← 1
while attempts < max_iterations and current_loss > 0 do
   best_neighbor ← None
   best_neighbor_loss ← ∞
   best_move ← None
   for each node in nodes(G) do
      for each color in range(num_colors) do
         if color ≠ current[node] then
            move ← (node, current[node], color)
            if move ∉ tabu_list then
               neighbor ← current.copy()
               neighbor[node] ← color
               neighbor_loss ← calculate_loss(G, neighbor)
               attempts ← attempts + 1
               if neighbor_loss < best_neighbor_loss then
                  best_neighbor ← neighbor
                  best_neighbor_loss ← neighbor_loss
                  best_move ← move
               end if
            end if
         end if
      end for
   end for
   if best_neighbor ≠ None then
      current ← best_neighbor
      current_loss ← best_neighbor_loss
      if current_loss < best_loss then
         best_solution ← current
         best_loss ← current_loss
      end if
      tabu_list ← tabu_list + [best_move]
      if length of tabu_list ¿ tabu_size then
         tabu_list.pop(0)
      end if
   else
      BREAK
   end if
end while
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

$num\_colors \leftarrow \frac{|nodes(G)|}{2}$

$current \leftarrow generate\_random\_coloring(G, num\_colors)$

$current\_loss \leftarrow calculate\_loss(G, current)$

$best\_solution \leftarrow current$

$best\_loss \leftarrow current\_loss$

$temperature \leftarrow initial\_temp$

$attempts \leftarrow 1$

**while** $temperature > min\_temp$ and $attempts < max\_iterations$ and $current\_loss > 0$ **do**

    $neighbor \leftarrow get\_gaussian\_neighbor(G, current, num\_colors)$

    $neighbor\_loss \leftarrow calculate\_loss(G, neighbor)$

    $attempts \leftarrow attempts + 1$

    $delta \leftarrow neighbor\_loss - current\_loss$

    **if** $delta < 0$ or random() ¡ exp(-delta / temperature) **then**

        $current \leftarrow neighbor$

        $current\_loss \leftarrow neighbor\_loss$

        **if** $current\_loss < best\_loss$ **then**

            $best\_solution \leftarrow current$

            $best\_loss \leftarrow current\_loss$

        **end if**

    **end if**

    $temperature \leftarrow get\_temperature(initial\_temp, attempts, max\_iterations, schedule)$

**end while**

**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

$num\_colors \leftarrow \frac{|nodes(G)|}{2}$
$population \leftarrow generate\_initial\_population(G, population\_size, num\_colors)$
$attempts \leftarrow population\_size$
$best\_solution \leftarrow max(population, keyfitness)$
$generation \leftarrow 0$
**while** termination condition not met **do**
  Sort $population$ by fitness in descending order
  **if** $best\_solution.fitness = 0$ **then**
    **return** $best\_solution.coloring, 0, attempts$
  **end if**
  **if** termination type is generations and $generation \geq max\_generations$ **then**
    **return** $best\_solution.coloring, -best\_solution.fitness, attempts$
  **end if**
  $new\_population \leftarrow population[: elite\_size]$
  **while** size of $new\_population < population\_size$ **do**
    $parent1, parent2 \leftarrow selectparentsfromtophalfofpopulation$
    **if** $crossover\_type = CrossoverType.UNIFORM$ **then**
      $child1, child2 \leftarrow uniform\_crossover(parent1, parent2)$
    **else**
      $child1, child2 \leftarrow single\_point\_crossover(parent1, parent2)$
    **end if**
    **if** $mutation\_type = MutationType.RANDOM$ **then**
      $child1 \leftarrow random\_mutation(child1, num\_colors)$
      $child2 \leftarrow random\_mutation(child2, num\_colors)$
    **else**
      $child1 \leftarrow swap\_mutation(child1)$
      $child2 \leftarrow swap\_mutation(child2)$
    **end if**
    $fitness1 \leftarrow -calculate\_loss(G, child1)$
    $fitness2 \leftarrow -calculate\_loss(G, child2)$
    $attempts \leftarrow attempts + 2$
    Add $child1, child2$ to $new\_population$
    **if** $fitness1 = 0$ or $fitness2 = 0$ **then**
      **return** best solution with fitness 0, $best\_solution.coloring, 0, attempts$
    **end if**
  **end while**
  Update population to $new\_population$
  $best\_solution \leftarrow max(population, keyfitness)$
  $generation \leftarrow generation + 1$
**end while**
**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

$num\_colors \leftarrow \frac{|nodes(G)|}{2}$
$attempts \leftarrow 0$
**if** $num\_processes$ is None **then**
  $num\_processes \leftarrow cpu\_count()$
**end if**
**INITIALIZE multiprocessing pool with** $num\_processes$
$coloring \leftarrow [generate\_random\_coloring(G, num\_colors)$ for $i = 1$ to $population\_size]$
$fitnesses \leftarrow evaluate\_population\_parallel(G, coloring, pool)$
$attempts \leftarrow attempts + population\_size$
$population \leftarrow [Individual(c, f)$ for $c, f$ in $zip(coloring, fitnesses)]$
$best\_solution \leftarrow \max(population, key = \lambda x : x.fitness)$
$generation \leftarrow 0$
**while** termination condition not met **do**
  SORT $population$ by fitness in descending order
  **if** $best\_solution.fitness = 0$ **then**
    **return** $best\_solution.coloring, 0, attempts$
  **end if**
  **if** $termination\_type = TerminationType.GENERATIONS$ and $generation \geq max\_generations$ **then**
    **return** $best\_solution.coloring, -best\_solution.fitness, attempts$
  **end if**
  $new\_population \leftarrow population[: elite\_size]$
  $offspring\_colorings \leftarrow []$
  **while** $|new\_population| + |offspring\_colorings|/2 < population\_size$ **do**
    $parent1, parent2 \leftarrow random.sample(population[: population\_size/2], 2)$
    **if** $crossover\_type = CrossoverType.UNIFORM$ **then**
      $child1, child2 \leftarrow uniform\_crossover(parent1, parent2)$
    **else**
      $child1, child2 \leftarrow single\_point\_crossover(parent1, parent2)$
    **end if**
    **if** $mutation\_type = MutationType.RANDOM$ **then**
      $child1 \leftarrow random\_mutation(child1, num\_colors)$
      $child2 \leftarrow random\_mutation(child2, num\_colors)$
    **else**
      $child1 \leftarrow swap\_mutation(child1)$
      $child2 \leftarrow swap\_mutation(child2)$
    **end if**
    $offspring\_colorings.append(child1, child2)$
  **end while**
  $offspring\_fitnesses \leftarrow evaluate\_population\_parallel(G, offspring\_colorings, pool)$
  $attempts \leftarrow attempts + |offspring\_fitnesses|$
  **for** $i = 0$ to $|offspring\_colorings| - 1$ step 2 **do**
    $child1\_coloring \leftarrow offspring\_colorings[i]$
    $child2\_coloring \leftarrow offspring\_colorings[i + 1]$
    $child1\_fitness \leftarrow offspring\_fitnesses[i]$
    $child2\_fitness \leftarrow offspring\_fitnesses[i + 1]$
    **if** $child1\_fitness = 0$ or $child2\_fitness = 0$ **then**
      $best\_coloring \leftarrow child1\_coloring$ if $child1\_fitness = 0$ else $child2\_coloring$

**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

```
num_colors ← |nodes(G)| / 2
attempts ← 0
island_size ← population_size / num_islands
if num_processes is None then
  num_processes ← cpu_count()
end if
INITIALIZE multiprocessing pool with num_processes
INITIALIZE islands with num_islands each having island_size individuals
for each island i do
  coloring ← [generate_random_coloring(G, num_colors) for i = 1 to island_size]
  fitnesses ← evaluate_population_parallel(G, coloring, pool)
  attempts ← attempts + island_size
  population ← [Individual(c, f) for c, f in zip(coloring, fitnesses)]
  islands.append(population)
end for
best_solution ← max(max(island, key = λx : x.fitness)foreachislandinislands)
generation ← 0
while termination condition not met do
  for each island i do
    SORT islands[i] by fitness in descending order
    current_best ← max(islands[i], key = λx : x.fitness)
    if current_best.fitness > best_solution.fitness then
      best_solution ← current_best
    end if
    if best_solution.fitness = 0 then

      return best_solution.coloring, 0, attempts
    end if
    if termination_type = TerminationType.GENERATIONS and generation ≥ max_generations then

      return best_solution.coloring, −best_solution.fitness, attempts
    end if
    new_population ← islands[i][: elite_size]
    offspring_colorings ← []
    while |new_population| + |offspring_colorings|/2 < island_size do
      parent1, parent2 ← random.sample(islands[i][: island_size/2], 2)
      if crossover_type = CrossoverType.UNIFORM then
        child1, child2 ← uniform_crossover(parent1, parent2)
      else
        child1, child2 ← single_point_crossover(parent1, parent2)
      end if
      if mutation_type = MutationType.RANDOM then
        child1 ← random_mutation(child1, num_colors)
        child2 ← random_mutation(child2, num_colors)
      else
        child1 ← swap_mutation(child1)
        child2 ← swap_mutation(child2)
      end if
      offspring_colorings.append(child1, child2)
```