Chapter 4 Hybrid Evolutionary Algorithm: A Case Study on Graph Coloring

吕志鹏

教授，博士生导师
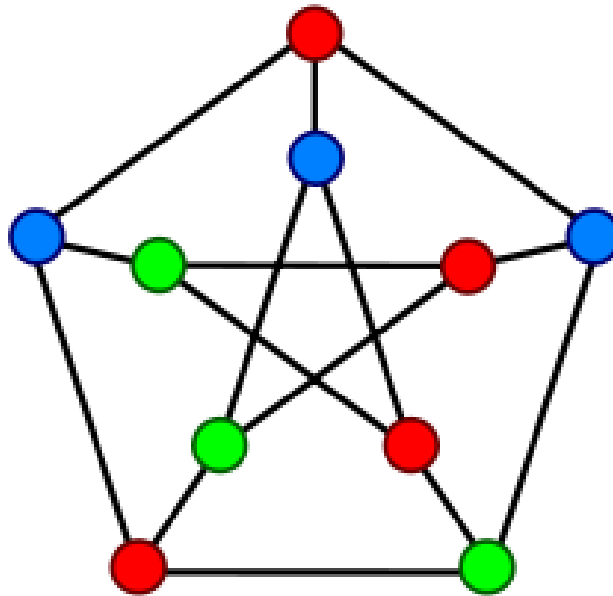
人工智能与优化研究所　所长

华中科技大学　计算机科学与技术学院

Part of this lecture has been given at the Dresden University of Technology in Germany

Graph Coloring

✳ Given an undirected graph G=(V,E), the graph coloring problem (GCP) consists of assigning a color c_i $(1 \leq c_i \leq k)$ to each vertex such that adjacent vertices receive different colors and the number of colors used k is minimized.

Optimization or Decision?

* GCP—**Optimization** Problem (NP Hard):
  * To find the smallest number of colors k.

* k-Coloring—**Decision** Problem (NP Complete):
  * Given a k, we are asked whether there exists a coloring such that all the adjacent coloring constraints are satisfied.

* The **Optimization** version of GCP can be solved by tackling a series of the **Decision** version of GCP problem with a gradually decreasing k.

* Thus, these two versions are equivalent to each other.

Solution Procedure

* We starts from an initial k and solve the k-coloring problem. As soon as the k-coloring problem is solved, we decrease k by setting k to k-1 and solve again the k-coloring problem.

* This process is repeated until no legal k-coloring can be found.

* Smaller k → harder k-coloring problem. Thus, the solution approach just described solves thus a series of k-coloring problems of increasing difficulty.

* We only consider the K-coloring problem in this presentation.

ILP Formulation

$$x_{v,c} = \begin{cases} 1 & \text{if vertex } v \text{ is coloured with colour } c \\ 0 & \text{otherwise} \end{cases}$$

$$\sum_{c=1}^{k} x_{v,c} = 1 \quad \forall \text{ vertices } v \in V$$

$$x_{u,c} + x_{v,c} \leq 1 \quad \forall \text{ colours } c \in K \quad \forall \text{ edges } \{u, v\} \in E$$

* Given **k** colors, $x_{v,c}$ is the decision variables.

* The first constraint requires that each vertex receives only one color.

* The second constraint denotes that adjacent vertices should receive different colors.

Assignment Representation

* A solution of k-coloring problem can be represented as a series of colors that each vertex receives:

* $S = \{c_1, c_2, \dots, c_n\}$ where c_v denotes the color of vertex *v*. It is required that for any $(u, v) \in E$, $c_u \neq c_v$.

* This representation is **natural,** but not **intuitive** and **essential.**

Grouping Representation

* The feasible solution of k-coloring problem can also be presented as a set of independent sets, where an independent set is a set of non-adjacent vertices.

* $S = I_1 \cup I_2 \cup \cdots \cup I_k$, where
    1. $I_j \cap I_l = \emptyset$ for any j and l
    2. $I_1 \cup I_2 \cup \cdots \cup I_k = V$
    3. I_j is an independent set for any j.

* Thus, the k-coloring problem becomes to partition the N vertices into k independent sets.

Applications

* Mobile radio frequency assignment

* Timetabling: Education, transportation, sports

* Register allocation

* Crew scheduling

* Printed circuit testing

* Air traffic flow management

* Satellite range scheduling

* Routing and wavelength assignment in WDM networks

Literature Review (1)

Constructive Greedy Algorithms

* The first heuristic approaches to solving the graph coloring problem, which color the vertices of the graph one by one guided by a greedy function.

* They are very fast by nature but their quality is unsatisfactory.

* The best known algorithms in this class are:
    * the *largest saturation degree* heuristic (DSATUR) (D. Brelaz, 1979)
    * *recursive largest first* heuristic (RLF) (F.T. Leighton, 1979)

* Often used to generate initial solutions for advanced algorithms

Literature Review (2)

Local Search Algorithms

* One representative example is the so-called *Tabucol* algorithm which is the first application of Tabu Search to graph coloring (Hertz, de Werra, 1987)

* Other local search metaheuristic methods include:
  * Simulated Annealing (Johnson et al, 1991)
  * Iterated Local Search (Chiarandini and Stutzle, 2002)
  * Reactive Partial Tabu Search (Blochliger, Zufferey, 2008)
  * GRASP (Laguna, Marti, 2001)
  * Variable Neighborhood Search (Avanthay et al, 2003)
  * Variable Space Search (Hertz et al, 2008)
  * *Clustering*-Guided Tabu Search (Porumbel, Hao, 2009)

* Interested readers are referred to [Galinier, Hertz, 2006] for a comprehensive survey of the local search approaches.

* I will describe the famous *Tabucol* algorithm in detail.

Literature Review (3)

Hybrid Evolutionary Algorithms

* One of the most recent and very promising approaches is based upon hybridization that embeds a **local search** algorithm into the framework of an **evolutionary algorithm** in order to achieve a better tradeoff between intensification and diversification, see for examples:

  * Dorne, Hao, 1998
  * Galinier, Hao, 1999
  * Galinier, Hertz, Zufferey, 2008
  * Malaguti, Monaci, Toth, 2008
  * Porumbel, Hao, Kuntz, 2009

Initial Solution——DSATUR

*The heuristic of DSATUR(Degree of Saturation) is to sequentially color vertices according to a DANGER-based heuristic. The main idea of DSATUR is based on least saturation degree.

*At each phase, it consists of two steps: The first is to choose a vertex to color and the other is to choose a color for the chosen vertex.

Initial Solution——DSATUR

* DSATUR starts by assigning color 1 to a vertex of maximal degree.

* Suppose F is a partial coloring of the vertices of G. The degree of saturation of a vertex x, degs(x), is the number of available colors that vertex x can use.

* The vertex to be colored next in the sequential coloring procedure of DSATUR is a vertex x with smallest degs(x), breaking ties by favoring vertex with larger uncolored degree.

* When deciding a color for a chosen vertex the color that is least likely to be required by neighboring vertices is selected.

DSATUR

1. Initialization:

* Color[N] = -1
* Vetex_Color_Avail[N][K]=1
* Num_Avail_Colors[N]=K
* Vertex_Uncolored_Degree[N]=Degree[K]

2. Choose the vertex v1 with the largest degree, color v1 with color 1

    Color[v1] = 1 ;

    for v1's adjacent vertices vj

        Vetex_Color_Avail[vj][1] = 0 ;

        Num_Avail_Colors[vj] -- ;

        Vertex_Uncolored_Degree[vj] -- ;

DSATUR

3. for(i = 2 ; i <= N ; i ++)

  * 3.1 choose a vertex vi according to:
  * < Num_Avail_Colors[vi] (small), Vertex_Uncolored_Degree[vi](large)>
  * 3.2 choose a color ki for vertex vi:
  * for each available color j for vertex vi
  * calculate the number of uncolored vertices for which color j is available
  * choose color ki with the smallest value of this number
  * 3.3 color vertex vi with color ki and updating:
  * Color[vi] = ki ;
    for vi's adjacent vertices vj
        Vertex_Uncolored_Degree[vj] -- ;
        if(Vetex_Color_Avail[vj][ki] == 1)
            Num_Avail_Colors[vj] -- ;
            Vetex_Color_Avail[vj][ki] = 0 ;

Improvements for DSATUR

* It is possible to improve DSATUR heuristic by considering more sophisticated information.

* For example, in case of choosing a color for the vertex, it would be better to consider the number of available colors.

* What else heuristics can be inspired in choosing vertex and choosing color?

Local Search

TabuCol

Search Space

* In this paper, we adapt the **k-fixed penalty strategy** which is also used by many coloring algorithms.

* For a given graph $G = (V; E)$, the number k of colors is fixed and the search space contains all possible (legal and illegal) k-colorings.

* A k-coloring is represented by $S = \{V_1, \ldots, V_k\}$ such that V_i is the set of vertices receiving color i.

* Thus, if for all V_i *are independent sets*, then S is a legal k-coloring. Otherwise, S is an illegal (or conflicting) k-coloring.

Evaluation Function

* The optimization objective is then to minimize the number of conflicting edges (referred to *confict number* hereafter) and find a legal *k-coloring* in the search space.

* Given a *k-coloring S = {V1,...,Vk}*, the evaluation function *f* counts the conflict number induced by *S* such that

$$f(S) = \sum_{\{u,v\} \in E} \delta_{uv}$$

where

$$\delta_{uv} = \begin{cases} 1, \text{ if } u \in V_i, \ v \in V_j \text{ and } i = j, \\ 0, \text{ otherwise.} \end{cases}$$

Initial Coloring

* The initial solution of our algorithm is randomly generated, i.e., each vertex in the graph is randomly assigned a color from 1 to k.

* Other greedy constructive heuristics are possible, like DSATUR, RLF, DANGER, etc.

* However, we observe that strong local search algorithms are not sensitive to the initial solutions.
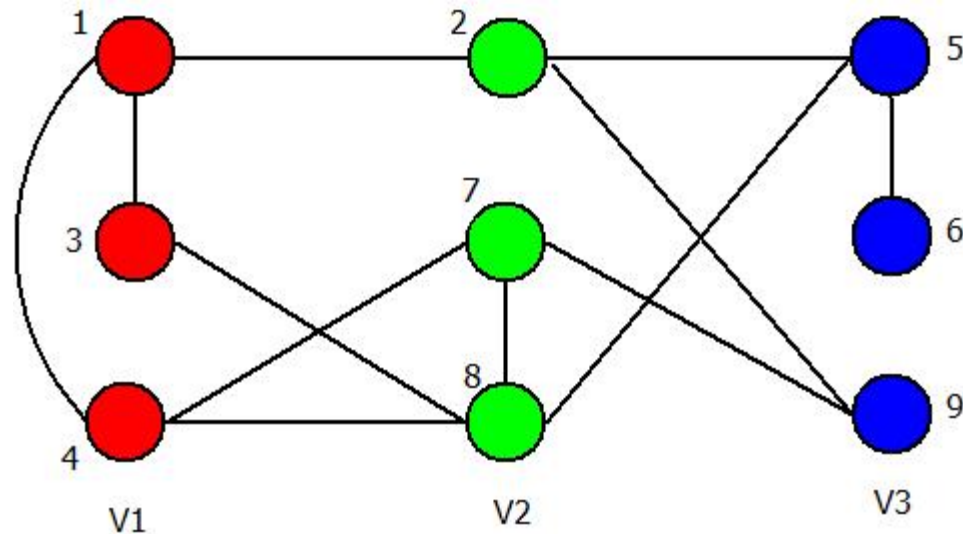
Neighborhood Moves

* A neighborhood of a given *k*-coloring is obtained by moving a ***conflicting*** vertex *u* from its original color class V_i to another color class V_j (denoted by *<u, i, j>*), called "critical one-move" neighborhood. A vertex is ***conflicting*** means that at least one of its adjacent vertices has the same color.

* Therefore, for a *k*-coloring *S* with cost *f(S)*, the size of this neighborhood is bounded by $O(f(S) \times k)$.

An Example

* f = 4

* Conflicting pairs: (1,3), (1,4), (7,8), (5,6)

* Critical One-Move: Only considers vertices 1, 3, 4, 7, 8, 5, 6. Totally 7*2=14 moves.

Neighborhood Evaluation

* In order to evaluate the neighborhood efficiently, we employ an *incremental evaluation* technique.

* The effect of each move on the objective function can be quickly calculated by a special data structure.

* Each time a move is carried out, only the move values affected by this move are updated accordingly.

Adjacent-Color Table

Vertex	Red V₁	Green V₂	Blue V₃
1	**2**	1	0
2	1	**0**	1
3	**1**	1	0
4	**1**	2	0
5	0	2	**1**
6	0	0	**1**
7	1	**1**	1
8	2	**1**	1
9	0	2	**0**
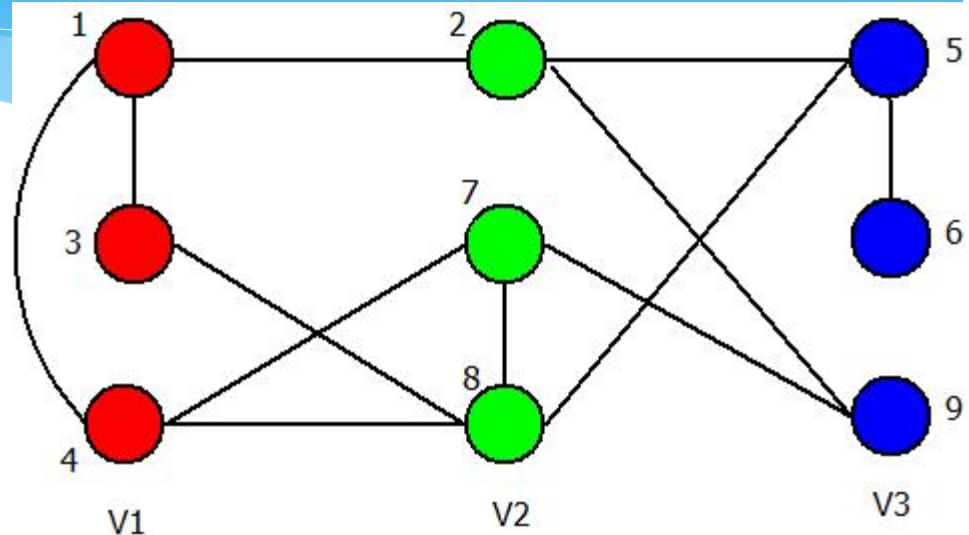
※ This matrix M[u][i] (N*k) measures the number of adjacent vertices if vertex **u** receives color **i**.

∗ Thus, the incremental move value of a move <u, i, j> can be quickly calculated as:

$$\Delta(u, i, j) = M[u][j] - M[u][i]$$

Updating of Adjacent-Color Table

Vertex	Red V1	Green V2	Blue V3
1	2	**1**	0
2	1-1=0	**0+1=1**	1
3	**1-1=0**	1+1=2	0
4	**1-1=0**	2+1=3	0
5	0	2	1
6	0	0	**1**
7	1	**1**	1
8	2	**1**	1
9	0	2	**0**
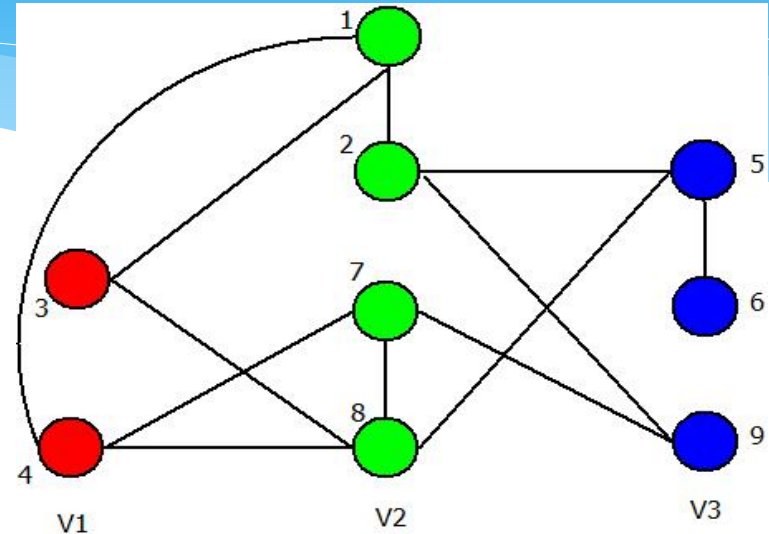
* Move (1, v1, v2):

* Only its adjacent vertices 2, 3 and 4 are affected, and only the v1 and v2 columns need to be updated.

* All old color (v1) columns decrease by 1.
* All new color (v2) columns increase by 1.

Simple Local Search

1. Generate initial solution S, Calculate f(S)

2. Initialize the adjacent-color table M.

3. While {there exist improving moves}

3.1   Construct the neighborhood of S, denoted by N(S)

3.2   Calculate the Δ values of all critical one-moves

3.3   Find the best move with the least Δ value

3.4   Perform the best move: $f' = f + \Delta_{best}$

3.5   Update the adjacent-color table M

    End

Tabu Search
Escaping from Local Optimum

* Tabu Search incorporates a tabu list as a "recency-based" memory structure to assure that solutions visited within a certain span of iterations, called tabu tenure, will not be revisited.

* TS then restricts consideration to moves not forbidden by the tabu list, and selects a move that produces the best move value to perform.

Tabu Search

* 1. What?

* 2. Tenure?

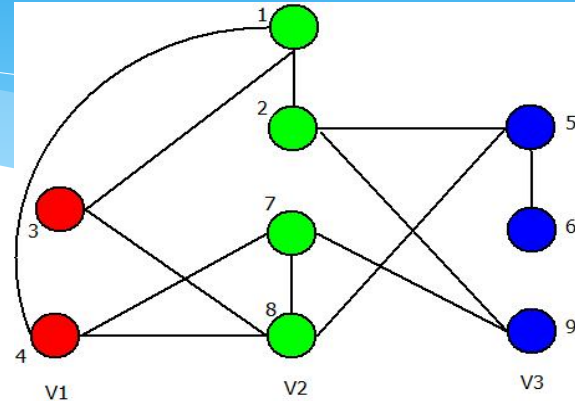* 3. How to judge if a move is forbidden?

Attributes or Solution?

* It should be noted that we generally forbid **attributes** of solutions, but not the **solutions** themselves, since it is too expensive to forbid **solutions**.

* For the tabu list, once move *<u, i, j>* is performed, vertex *u* is forbidden to move back to color class *Vi* for the next *tt* iterations.

Tabu Tenure

* For the tabu list, once move $<u, i, j>$ is performed, vertex u is forbidden to move back to color class V_i for the next tt iterations.

* Here, the tabu tenure tt is dynamically determined by

$$tt = f(S) + r(10)$$

where $r(10)$ takes a random number in $\{1,\dots,10\}$.

TabuTenure Table

Vertex	Red V1	Green V2	Blue V3
1	9	0	0
2	0	10	0
3	0	0	0
4	0	0	0
5	0	0	0
6	0	0	0
7	0	0	0
8	0	0	0
9	0	0	0

* At the begining of the search, the TabuTenure table is initialized to be zero.
* once move *<u, i, j>* is performed, the value of Table[u][i]=TabuTenure.
* Once the search progresses, the non-zero value of the table is decreased by one at each time.
* In the following search, we can decide if a move *<u, i, j>* is tabu by checking if Table[u][j]>0.

Fast Implementation of TabuTenure Table

Vertex	Red V1	Green V2	Blue V3
1	1+10	0	0
2	0	2+15	0
3	0	0	0
4	0	0	3+12
5	0	0	0
6	0	0	0
7	0	4+10	0
8	0	0	0
9	0	0	5+12
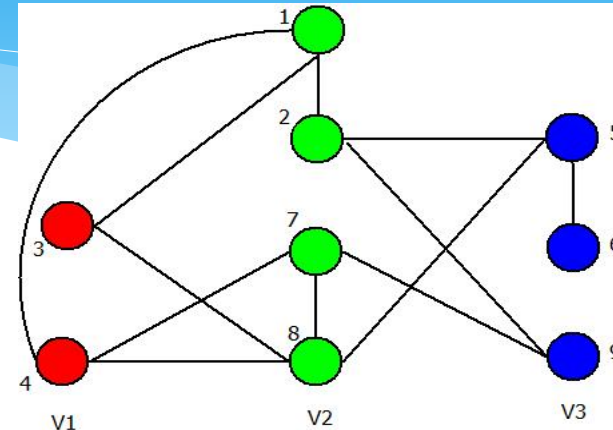
* The above Table[u][i] records the relative tabu tenure length. Why not record the absolute tabu tenure length?

* Once move $<u, i, j>$ is performed, the value of Table[u][i] = TabuTenure+Iter.

* In the following search, we can decide if a move $<u, i, j>$ is tabu by checking if Table[u][j]>Iter.

* In this way, the tabu tenure table can be updated in O(1).

Aspiration

* If one move can override the best found solution found so far, it is accepted even if it is in tabu status.

* This is because only the **attributes** but not **solutions** themselves are stored in the tabu table.

TS Algorithm

1. Generate initial solution S, Calculate $f(S)$
2. Initialize the adjacent-color table M.
3. While {stop condition is not met}
3.1   Construct the neighborhood of S, denoted by N(S)
3.2   Calculate the Δ values of all critical one-moves
3.3   Find the best tabu and non-tabu moves with the least Δ value
3.4   If {the aspiration condition is satisfied}
             perform the best tabu move,
         else
             perform the best non-tabu move
3.5   Update f and the adjacent-color table M
    End

TS Algorithm

* Data Structures: Sol[N], f, BestSol[N], Best_f, TabuTenure[N][K], Adjacent_Color_Table[N][K]

* Subfunctions:

    * Initialization(): Initialize the values of the data structures.

    * FindMove(u,vi,vj,delt): find the best non-tabu or tabu move.

    * MakeMove(u,vi,vj,delt): update the correponding values.

* TabuSearch()

* { int u, vi, vj, iter = 0;

* Initialization() ;

* while(iter < MaxIter) {

*   FindMove(u,vi,vj,delt) ;

*   MakeMove(u,vi,vj,delt) ; }

* }

Find Move

* FindMove(u,vi,vj,delt)

* {

*   for(i=1:N)

*   if(Adjacent_Color_Table[i][Sol[i]] > 0) {

*   for(k = 1: K)

*   if(k != Sol[i]) {

*   calculate delt value of the move <i, Sol[i], k>

*   {

*   if (Table[i][k] < iter)   update the tabu best move;

*   else update the non-tabu best move;

    }}}

  if(the tabu best move satisfies the tabu aspiration criterion)

     <u, vi, vj, delt> = the tabu best move;

 else   <u, vi, vj,delt> = the non-tabu best move;

* }

Make Move

* MakeMove(u,vi,vj, delt)

* {

* Sol[u] = vj ;

* f = f + delt ;

* Table[u][vi] = iter + f + rand()%10;

* Update the Adjacent_Color_Table;

* }

Tips

* For both tabu and non-tabu moves, if there are multiple best moves, a random one is selected.

* Aspiration criterion: it holds if both the following two conditions satisfy:

* 1. the best tabu move is better than the previous best known solution;

* 2. the best tabu move is better than the best non-tabu move in the current neighborhood.

Hybrid Evolutionary Algorithm

Main Scheme
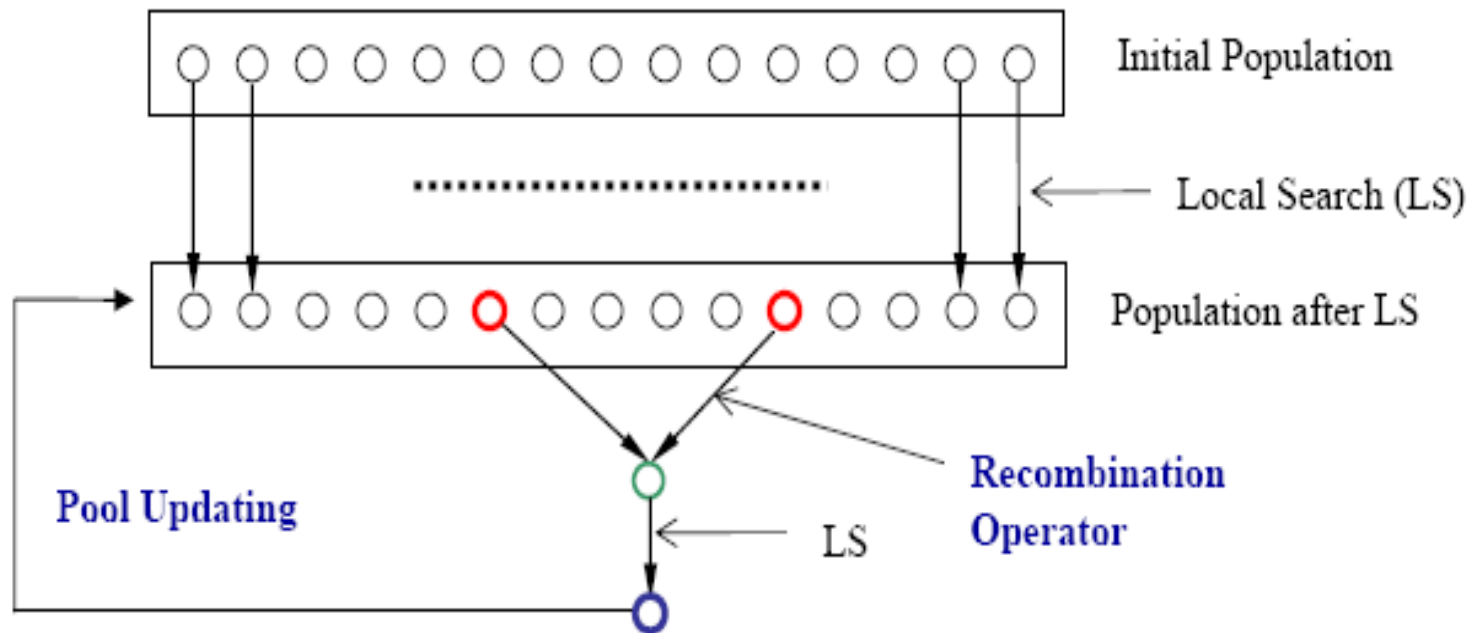
Hybrid Evolutionary Algorithm (LS+EA)

Hybrid Evolutionary Algorithm

The hybrid coloring algorithm

Data : $graph\ G = (V, E),\ integer\ k$
Result : $the\ best\ configuration\ found$
begin
    P=$InitPopulation(|P|)$
    while $not\ Stop\text{-}Condition\ ()$ **do**
        (s1,s2)=$ChooseParents$(P)
        s=$Crossover$(s1,s2)
        s=$LocalSearch$(s, L)
        P=$UpdatePopulation$(P,s)
end

Crossover Operator (1)

Table 2. The crossover algorithm: an example.

parent s_1 →	A B C	D E F G	H I J
parent s_2	C D E G	A F I	B H J
offspring s			

$V_1 := \{D, E, F, G\}$
remove D,E,F and G

	A B C		H I J
	C	A I	B H J
	D E F G		

parent s_1	A B C		H I J
parent s_2 →	C	A I	B H J
offspring s	D E F G		

$V_2 := \{B, H, J\}$
remove B,H and J

	A C		I
	C	A I	
	D E F G	B H J	

parent s_1 →	A C		I
parent s_2	C	A I	
offspring s	D E F G	B H J	

$V_3 := \{A, C\}$
remove A and C

			I
		I	
	D E F G	B H J	A C

Crossover Operator (2)

* A legal *k*-coloring is a collection of *k* independent sets.

* With this point of view, if we could maximize the size of the independent sets by a crossover operator as far as possible, it will in turn help to push those left vertices into independent sets.

* In other words, the more vertices are transmitted from parent individuals to the offspring within *k* steps, the less vertices are left unassigned.

* In this way, the obtained offspring individual has more possibility to become a legal coloring.

Crossover Operator (3)

The GPX crossover algorithm

Data : $configurations \ s_1 = \{V_1^1, \ldots, V_k^1\} \ and \ s_2 = \{V_1^2, \ldots, V_k^2\}$
Result : $configuration \ s = \{V_1, \ldots, V_k\}$
begin
    for $l(1 \leq l \leq k)$ **do**
        if l is odd, then $A := 1$, else $A := 2$
        choose i such that V_i^A has a maximum cardinality
        $V_l := V_i^A$
        remove the vertices of V_l from s_1 and s_2
    Assign randomly the vertices of $V - (V_1 \cup \cdots \cup V_k)$
end

HEA Algorithm Scheme

Algorithm 1 Pseudcode of the Hybrid Evolutionary Algorithm for k-Coloring

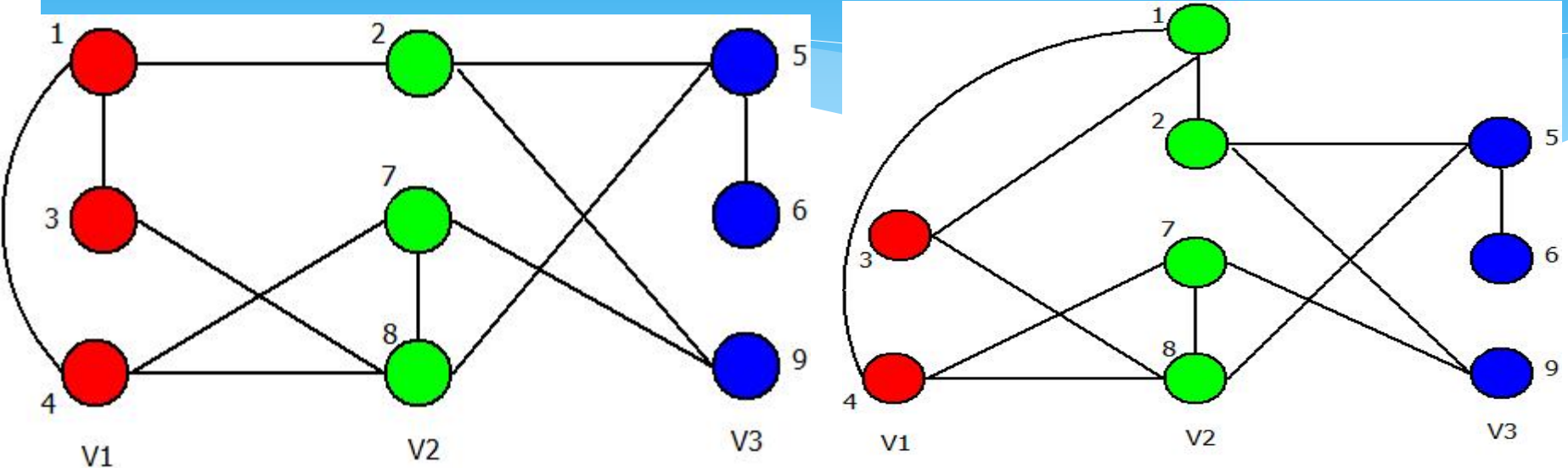1: **Input**: Graph G
2: **Output**: The best solution S^* found so far
3: $\{S_1, \ldots, S_p\} \leftarrow$ Initial Population
4: **for** $i = \{1, \ldots, p\}$ **do**
5:     $S_i \leftarrow$ Tabu Search(S_i)
6: **end for**
7: $S^* = arg\ min\{f(S_i), i = 1, \ldots, p\}$
8: **repeat**
9:     Randomly choose two parent solutions $\{S_{i1}, S_{i2}\}$
10:     $S_0 \leftarrow$ Crossover_Operator (S_{i1}, S_{i2})
11:     $S_0 \leftarrow$ Tabu Search(S_0)
12:     **if** $f(S_0) < f(S^*)$ **then**
13:         $S^* = S_0$
14:     **end if**
15:     $\{S_1, \ldots, S_p\} \leftarrow$ Pool Updating$(S_0, S_1, \ldots, S_p)$
16: **until** Stop condition met

图着色的高级玩法

更新冲突节点

* Move (1, v1, v2): what are the new conflicting vertices?

* Vertices 3 and 4 becomes un-conflicting, while vertex 2 becomes conflicting

* Use an array to store the conflicting vertices and update the array after each move

* Use Array to implement it with O(1) time complexity

更新冲突节点

* Num: 冲突节点数
* Con[N]：冲突节点数组，只有前Num有效
* Pos[N]：记录任意一个节点在Conf数组中的位置

* 加节点V：Conf[Num]＝V; Pos[V]=Num＋＋;
* 删节点V：Conf[Pos[V]]＝Conf[Num];
*   Pos[Conf[Num]]=Pos[V];
*   Num－－;

最优邻域动作如何快速找出来?
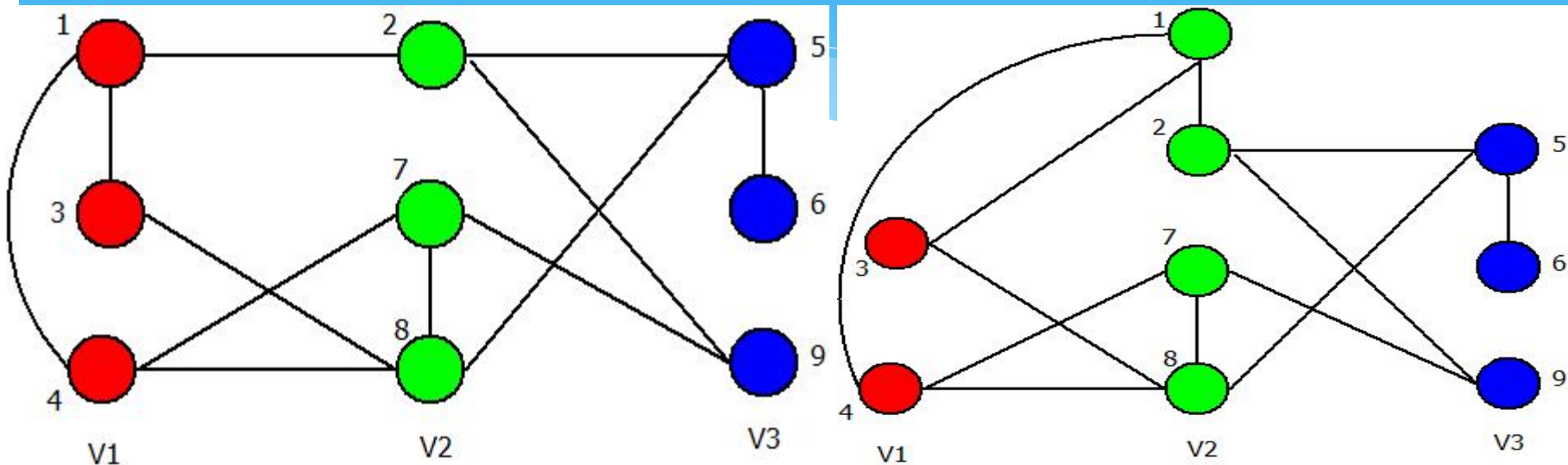
* 邻域大小: m*(k-1), m为冲突节点数, k为颜色数 (冲突节点维护技术: N->m)

* 邻域动作时只选择禁忌最好的或者非禁忌最好的

* 上述动作后, 冲突节点减少2个, 增加1个, 原来14个邻域动作, 现在变为12个

* 哪些邻域动作发生了变化? 哪些动作的delt值发生了变化?

最优邻域动作如何快速找出来?
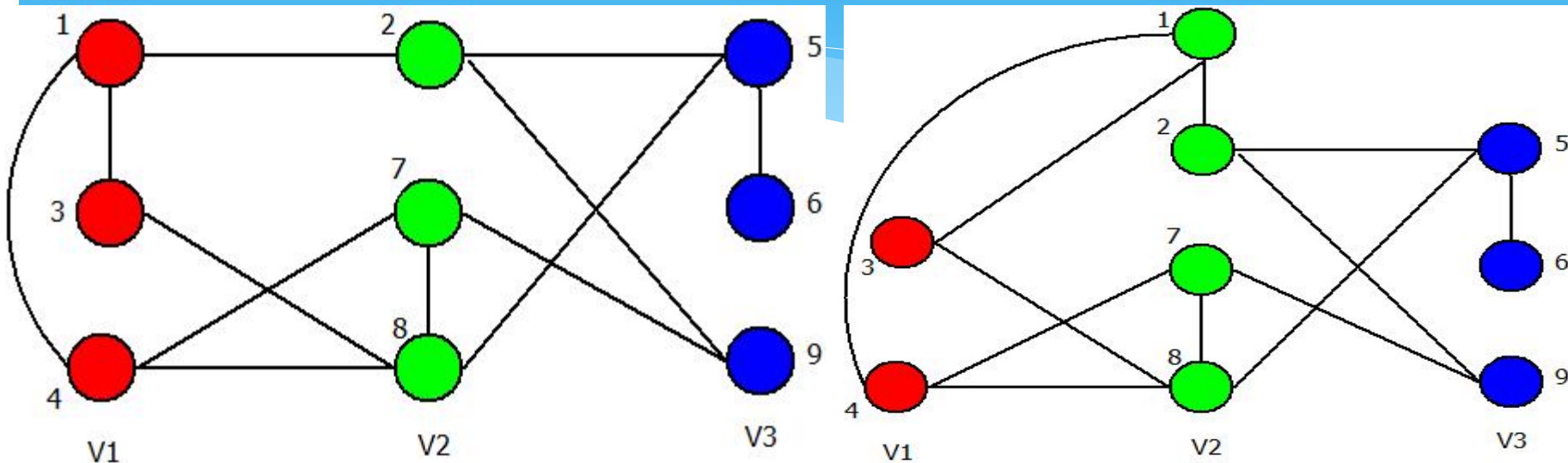
* 1. 减少的邻域动作: 节点3, 4到绿色、蓝色;
* 2. 增加的领域动作: 节点2到红色、蓝色;
* 3. delt值发生变化的领域动作: 无。

* 如果节点7与节点1相连, 那么节点7移到红色的仇人数－1, 移到绿色+1。即节点1的邻居节点如果仍为冲突节点, 它到红色的仇人数－1, 移到绿色+1。

* 既然大多数动作的delt值没有发生变化, 那么最优的delt值也有可能未发生变化

最优邻域动作如何快速找出来？
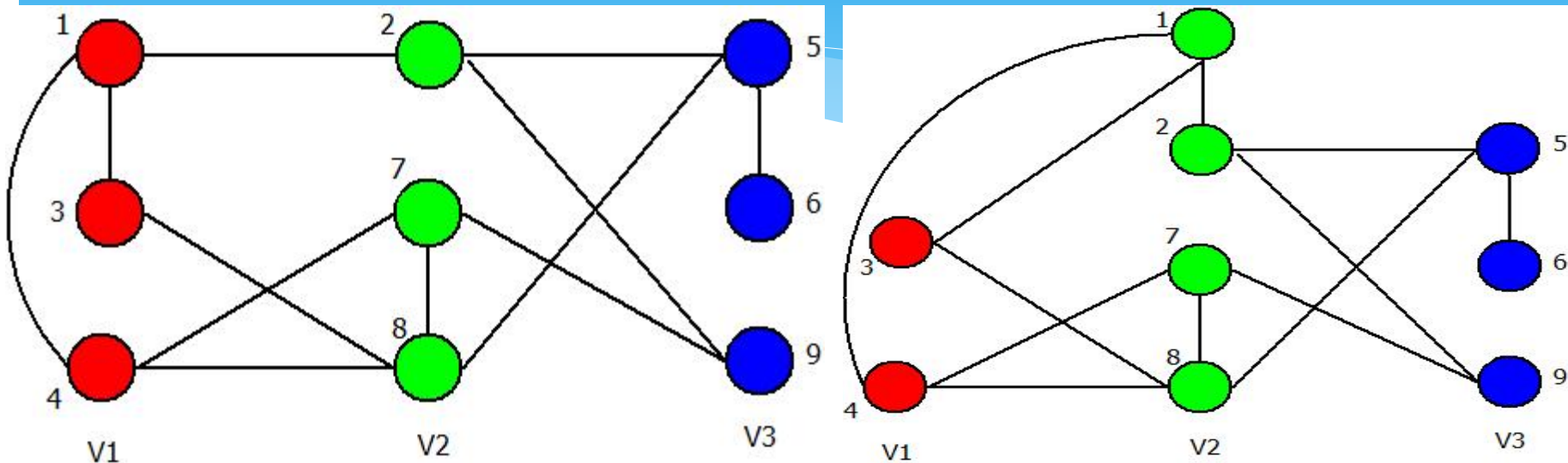
* 1. 减少的邻域动作直接删除
* 2. 增加的领域动作重新计算并加入
* 3. delt值发生变化的领域动作（即移动节点的邻居），分三种情况
*   a) 如果该节点在老颜色中，移到新颜色delt值+2，移到其它颜色delt值+1。
*   b) 如果该节点在新颜色中，移到老颜色delt值-2，移到其它颜色delt值-1。
*   c) 如果该节点在其它颜色中，移到老颜色delt值-1，移到新颜色delt值+1，其它不变

最优邻域动作如何快速找出来？

* 桶排序技术（增、删、移动与之前的数据实现冲突节点维护一致）：
* 辅助数据结构：Pos[N][k][k][2]

id	Delt值	邻域动作数	邻域动作（及delt值）
1	<=—4	0	
2	—3	0	
3	—2	4	……………..
4	—1	1	…
5	0	10	…………………………..
6	1	12	…………………………..
7	2	12	…………………………..
8	3	25	…………………………..
9	>=4	30	…………………………..

最优邻域动作如何快速找出来？

* 每次从动作数非0的第一个桶中随机选一个动作

* 桶更新策略：
    1. 删掉的邻域动作要从桶中删掉；
    2. 增加的邻域动作要加到桶中；
    3. delt值发生变化的动作要在桶中移动到对应的桶

该策略可以节约多少时间：
    邻域动作大小：m*(k-1)
    更新的邻域动作数:(增、删冲突节点数)*(k-1)+移动节点的邻居节点仍为冲突节点数 * 2

最优邻域动作如何快速找出来?

* 可以更懒吗?

* 惰性更新桶的策略:
    1. 应该删除的邻域动作不用管
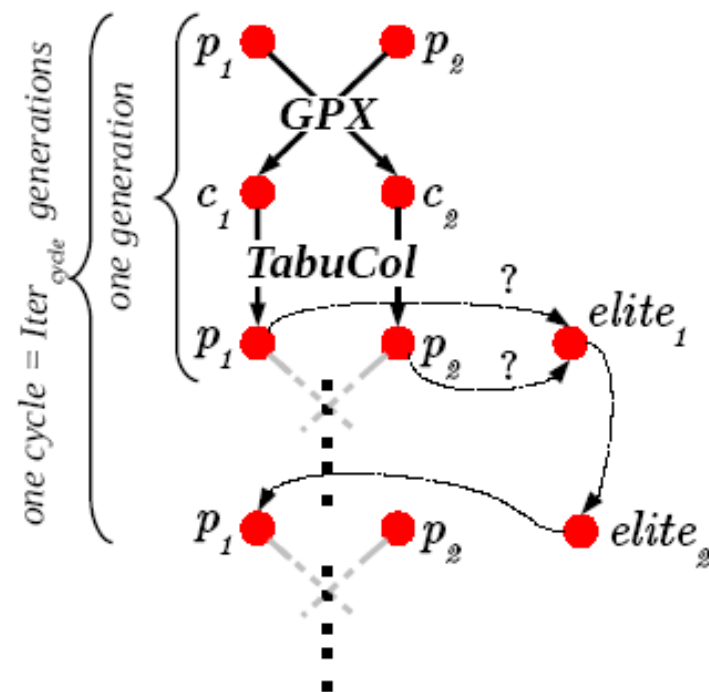   2. 新增加的邻域动作直接加到桶中;
    3. delt值发生变化的动作只在桶中加新的动作，旧动作不动

师徒混合进化算法

Fig. 2: Diagram of *HEAD*

师徒混合进化算法

Algorithm 2: $HEAD$ - second version of $HEAD$ with two extra elite solutions

Input: k, the number of colors; $Iter_{TC}$, the number of $TabuCol$ iterations; $Iter_{cycle} = 10$, the number of generations into one cycle.

Output: the best k-coloring found: $best$

1 $p_1, p_2, elite_1, elite_2, best \leftarrow$ init()   /* initialize with random k-colorings */

2 $generation, cycle \leftarrow 0$

3 **do**

4       $c_1 \leftarrow GPX(p_1, p_2)$

5       $c_2 \leftarrow GPX(p_2, p_1)$

6       $p_1 \leftarrow TabuCol(c_1, Iter_{TC})$

7       $p_2 \leftarrow TabuCol(c_2, Iter_{TC})$

8       $elite_1 \leftarrow$ saveBest($p_1, p_2, elite_1$) /* best k-coloring of the current cycle */

9       $best \leftarrow$ saveBest($elite_1, best$)

10   **if** $generation \% Iter_{cycle} = 0$ **then**

11           $p_1 \leftarrow elite_2$ /* best k-coloring of the previous cycle */

12           $elite_2 \leftarrow elite_1$

13           $elite_1 \leftarrow$ init()

14           $cycle + +$

15       $generation + +$

16 **while** $nbConflicts(best) > 0$ and $p_1 \neq p_2$

高级局部搜索算法

* 局部搜索算法500.5能算到48种颜色吗?

* 500.5能算够非常稳定地算到47种颜色吗?

* 局部搜索算法500.5能算到47种颜色吗?

* 禁忌算法的局限和新实现：格局检测

* 节点是否可以回到老颜色应该取决于老颜色中它的"格局"是否发生了变化

* 仅"格局检测"是不够的，还需要加权和平滑技术的配合使用

Thank You!