



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

吕志鹏

教授，博士生导师

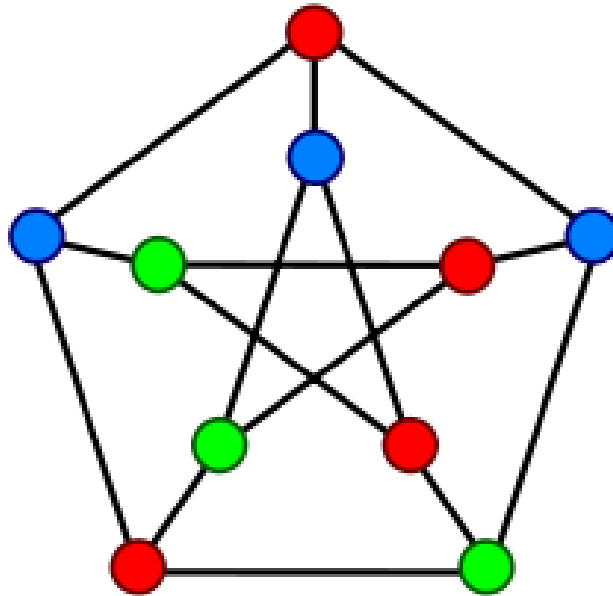
计算科学理论研究所 常务副所长

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 start 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  $k \rightarrow$  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.
- \*  $S = I_1 \cup I_2 \cup \dots \cup I_k$ , where
  1.  $I_j \cap I_l = \emptyset$  for any  $j$  and  $l$
  2.  $I_1 \cup I_2 \cup \dots \cup I_k = V$
  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  $x$ ,  $\text{deg}_s(x)$ , 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  $\text{deg}_s(x)$ , 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( $i = 2 ; i \leq N ; i ++$ )
  - \* 3.1 choose a vertex  $v_i$  according to:
    - \*  $\langle \text{Num\_Avail\_Colors}[v_i] (\text{small}), \text{Vertex\_Uncolored\_Degree}[v_i](\text{large}) \rangle$
  - \* 3.2 choose a color  $k_i$  for vertex  $v_i$ :
    - \* for each available color  $j$  for vertex  $v_i$
    - \* calculate the number of uncolored vertices for which color  $j$  is available
    - \* choose color  $k_i$  with the smallest value of this number
  - \* 3.3 color vertex  $v_i$  with color  $k_i$  and updating:
    - \*  $\text{Color}[v_i] = k_i ;$   
for  $v_i$ 's adjacent vertices  $v_j$ 
      - $\text{Vertex\_Uncolored\_Degree}[v_j] -- ;$
      - if( $\text{Vertex\_Color\_Avail}[v_j][k_i] == 1$ )
        - $\text{Num\_Avail\_Colors}[v_j] -- ;$
        - $\text{Vertex\_Color\_Avail}[v_j][k_i] = 0 ;$

# 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  $S = \{V_1, \dots, V_k\}$  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 *conflict number* hereafter) and find a legal  $k$ -coloring in the search space.
- \* Given a  $k$ -coloring  $S = \{V_1, \dots, V_k\}$ , 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

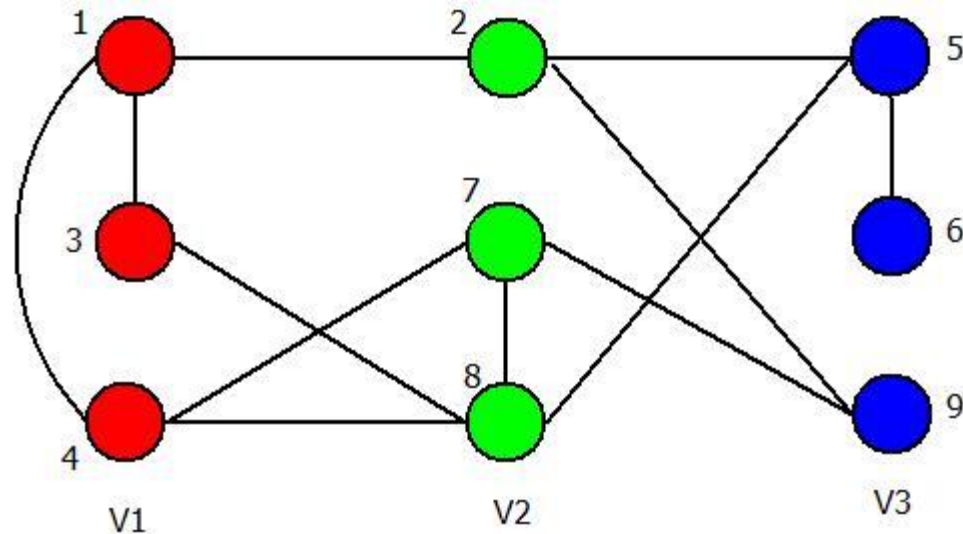
- \* A neighborhood of a given  $k$ -coloring is obtained by moving a **conflicting** vertex  $u$  from its original color class  $V_i$  to another color class  $V_j$  (denoted by  $\langle u, i, j \rangle$ ), called “**critical** one-move” neighborhood.
- \* Therefore, for a  $k$ -coloring  $S$  with 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.

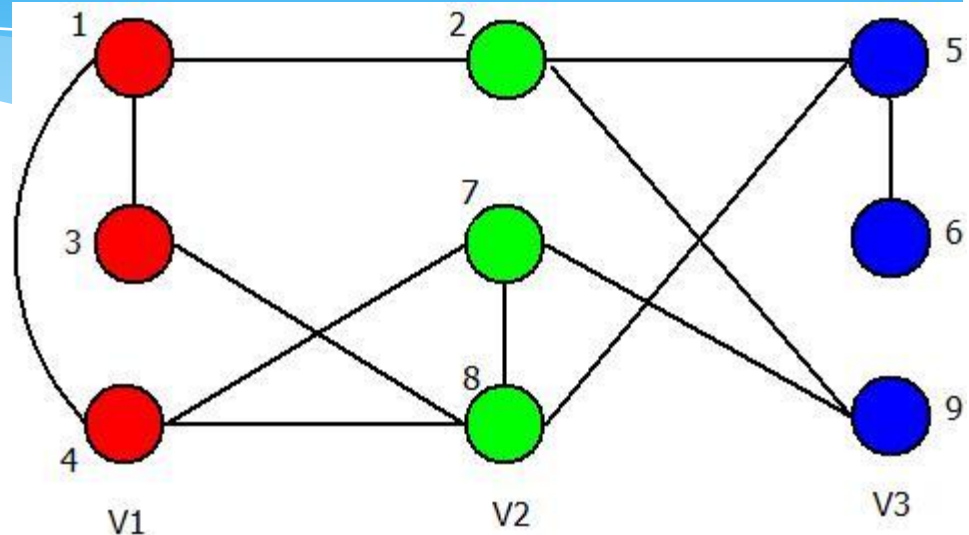


# Neighborhood Evaluation

- \* 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 V <sub>1</sub>	Green V <sub>2</sub>	Blue V <sub>3</sub>
1	<u>2</u>	1	0
2	1	<u>0</u>	1
3	<u>1</u>	1	0
4	<u>1</u>	2	0
5	0	2	<u>1</u>
6	0	0	<u>1</u>
7	1	<u>1</u>	1
8	2	<u>1</u>	1
9	0	2	<u>0</u>



\* This matrix  $M[u][i]$  ( $N \times k$ ) measures the number of adjacent vertices if vertex  $u$  receives color  $i$ .

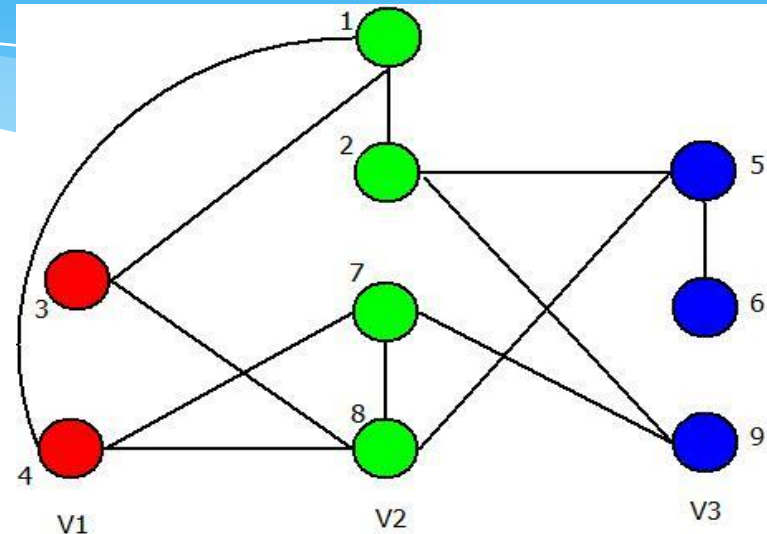
\* Thus, the incremental move value of a move  $\langle u, i, j \rangle$  can be quickly calculated as:

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



# Updating of Adjacent-Color Table

Vertex	Red V1	Green V2	Blue V3
1	2	<u>1</u>	0
2	$1-1=0$	$\underline{0+1=1}$	1
3	$\underline{1-1=0}$	$1+1=2$	0
4	$\underline{1-1=0}$	$2+1=3$	0
5	0	2	<u>1</u>
6	0	0	<u>1</u>
7	1	<u>1</u>	1
8	2	<u>1</u>	1
9	0	2	<u>0</u>



- \* 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  $\Delta$  values of all critical one-moves
  - 3.3 Find the best move with the least  $\Delta$  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?

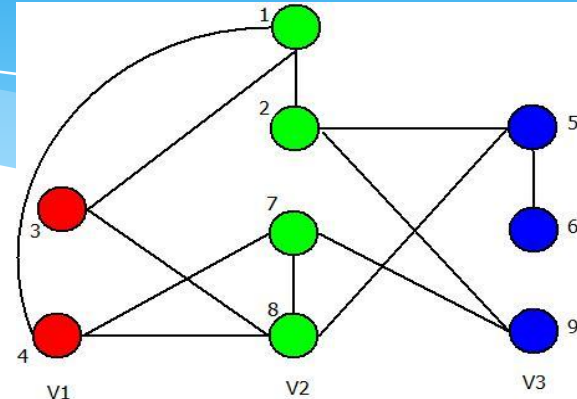
- \* It should be noted that we generally forbid **attributes** of solutions, but not the **solutions** themselves, since it is too expensive to forbid **solutions**.
- \* For the tabu list, once move  $\langle u, i, j \rangle$  is performed, vertex  $u$  is forbidden to move back to color class  $V_i$  for the next  $tt$  iterations.

# Tabu Tenure

- \* For the tabu list, once move  $\langle u, i, j \rangle$  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

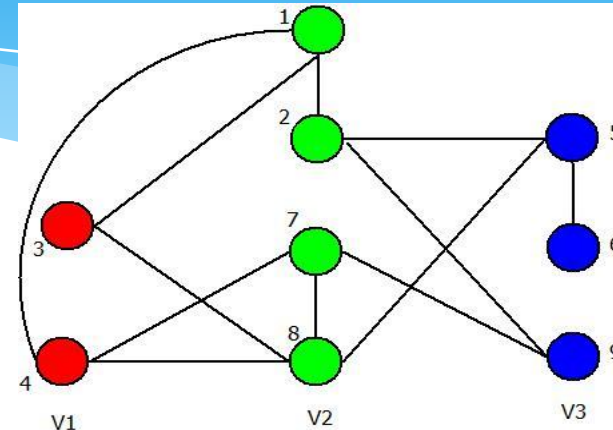
Vertex	Red V1	Green V2	Blue V3
1	9	0	0
2	0	10	0
3	0	0	0
4	0	0	0
5	0	0	0
6	0	0	0
7	0	0	0
8	0	0	0
9	0	0	0



- \* At the beginning of the search, the TabuTenure table is initialized to be zero.
- \* once move  $\langle u, i, j \rangle$  is performed, the value of  $\text{Table}[u][i] = \text{TabuTenure}$ .
- \* 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  $\langle u, i, j \rangle$  is tabu by checking if  $\text{Table}[u][j] > 0$ .

# 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  $\langle u, i, j \rangle$  is performed, the value of Table[u][i] = TabuTenure+Iter.
- \* In the following search, we can decide if a move  $\langle u, i, j \rangle$  is tabu by checking if Table[u][i] > 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  $\Delta$  values of all critical one-moves
  - 3.3 Find the best tabu and non-tabu moves with the least  $\Delta$  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 corresponding 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;
*       }
* }
```

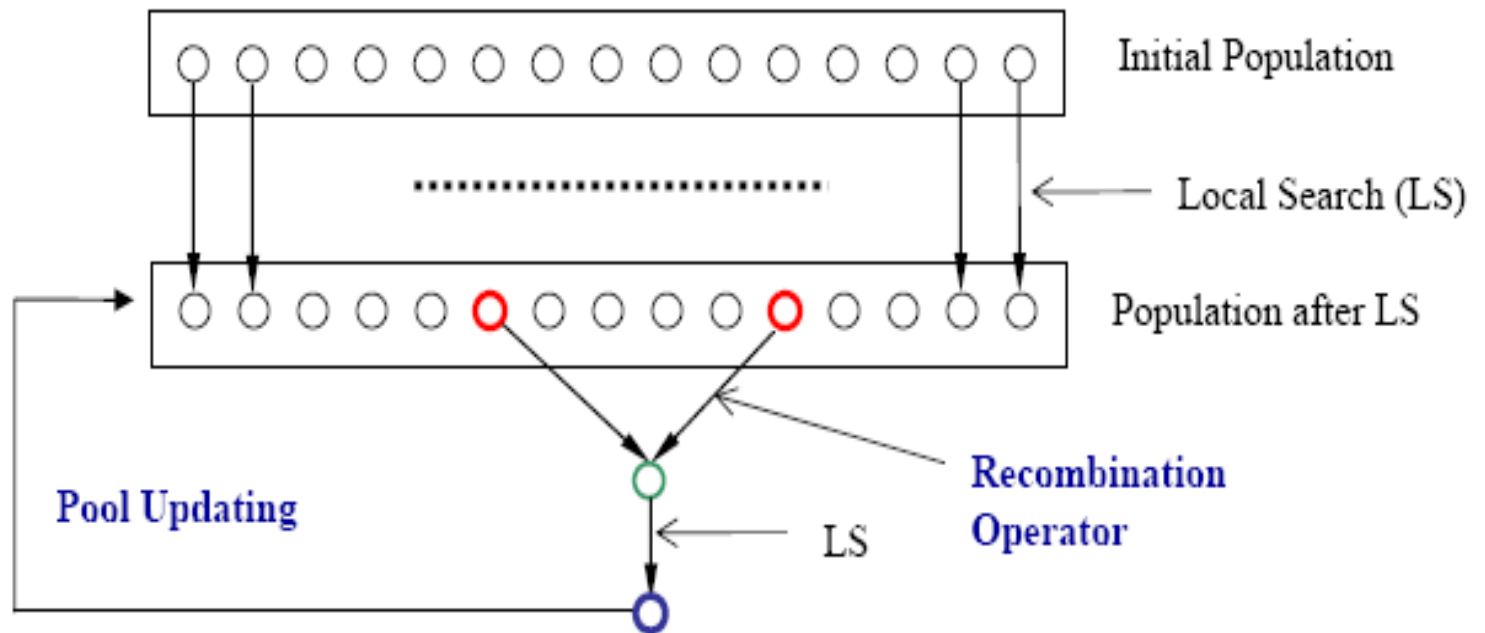
# Make Move

```
* MakeMove(u,vi,vj, delt)
* {
*   Sol[ u ] = vj;
*   f = f + delt;
*   Table[ u ][ vi ] = iter + f + rand()%10;
*   Update the Adjacent_Color_Table;
* }
```



# 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**

$P = \text{InitPopulation}(|P|)$

**while** *not Stop-Condition ()* **do**

$(s1, s2) = \text{ChooseParents}(P)$

$s = \text{Crossover}(s1, s2)$

$s = \text{LocalSearch}(s, L)$

$P = \text{UpdatePopulation}(P, s)$

**end**



# Crossover Operator (1)

Table 2. The crossover algorithm: an example.


parent $s_1 \rightarrow$	<table><tr><td><u>A B C</u></td><td><u>D E F G</u></td><td>H I J</td></tr></table>	<u>A B C</u>	<u>D E F G</u>	H I J	$V_1 := \{D, E, F, G\}$	<table><tr><td>A B C</td><td></td><td>H I J</td></tr></table>	A B C		H I J
<u>A B C</u>	<u>D E F G</u>	H I J							
A B C		H I J							
parent $s_2 \rightarrow$	<table><tr><td>C <u>D E G</u></td><td>A <u>F I</u></td><td>B H J</td></tr></table>	C <u>D E G</u>	A <u>F I</u>	B H J	remove D,E,F and G	<table><tr><td>C</td><td>A I</td><td>B H J</td></tr></table>	C	A I	B H J
C <u>D E G</u>	A <u>F I</u>	B H J							
C	A I	B H J							
offspring $s$	<table><tr><td></td><td></td><td></td></tr></table>					<table><tr><td>D E F G</td><td></td><td></td></tr></table>	D E F G		
D E F G									
parent $s_1 \rightarrow$	<table><tr><td><u>A B C</u></td><td></td><td><u>H I J</u></td></tr></table>	<u>A B C</u>		<u>H I J</u>	$V_2 := \{B, H, J\}$	<table><tr><td>A C</td><td></td><td>I</td></tr></table>	A C		I
<u>A B C</u>		<u>H I J</u>							
A C		I							
parent $s_2 \rightarrow$	<table><tr><td>C</td><td>A I</td><td><u>B H J</u></td></tr></table>	C	A I	<u>B H J</u>	remove B,H and J	<table><tr><td>C</td><td>A I</td><td></td></tr></table>	C	A I	
C	A I	<u>B H J</u>							
C	A I								
offspring $s$	<table><tr><td>D E F G</td><td></td><td></td></tr></table>	D E F G				<table><tr><td>D E F G</td><td>B H J</td><td></td></tr></table>	D E F G	B H J	
D E F G									
D E F G	B H J								
parent $s_1 \rightarrow$	<table><tr><td><u>A C</u></td><td></td><td>I</td></tr></table>	<u>A C</u>		I	$V_3 := \{A, C\}$	<table><tr><td></td><td></td><td>I</td></tr></table>			I
<u>A C</u>		I							
		I							
parent $s_2 \rightarrow$	<table><tr><td><u>C</u></td><td><u>A I</u></td><td></td></tr></table>	<u>C</u>	<u>A I</u>		remove A and C	<table><tr><td></td><td>I</td><td></td></tr></table>		I	
<u>C</u>	<u>A I</u>								
	I								
offspring $s$	<table><tr><td>D E F G</td><td>B H J</td><td></td></tr></table>	D E F G	B H J			<table><tr><td>D E F G</td><td>B H J</td><td>A C</td></tr></table>	D E F G	B H J	A C
D E F G	B H J								
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  $s_1 = \{V_1^1, \dots, V_k^1\}$  and  $s_2 = \{V_1^2, \dots, V_k^2\}$  

**Result :** configuration  $s = \{V_1, \dots, V_k\}$

**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  $V - (V_1 \cup \dots \cup V_k)$  

**end**

# HEA Algorithm Scheme

---

**Algorithm 1** Pseudocode of the Hybrid Evolutionary Algorithm for  $k$ -Coloring

---

```
1: Input: Graph  $G$ 
2: Output: The best solution  $S^*$  found so far
3:  $\{S_1, \dots, S_p\} \leftarrow$  Initial Population
4: for  $i = \{1, \dots, p\}$  do
5:    $S_i \leftarrow$  Tabu Search( $S_i$ )
6: end for
7:  $S^* = \arg \min\{f(S_i), i = 1, \dots, p\}$ 
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:   $\{S_1, \dots, S_p\} \leftarrow$  Pool Updating( $S_0, S_1, \dots, S_p$ )
16: until Stop condition met
```

---



**Thank You!**