图着色算法的编码、设计与分析

Tianjyan

日期：2020 年 6 月 30 日

摘　要

本文选择了图着色问题，对这个问题进行了 SAT 和 ILP 编码。然后，作者使用了三种算法，分别是结构化搜索，贪心局部搜索和模拟退火方法进行叙述，并分别给出了代码实现。最后，作者构建了一些测试用例，分别对这几种算法进行实验，并得出相应的结论。

1　问题描述

给定一个 n 个点和 m 条边无向图 G(V, E) 和 k 种颜色。图着色问题的判定版本是指，要求每个点染一种颜色，问是否存在一组染色方案，使得图中所有相邻的点的颜色互不相同。如果不存在符合要求的染色方案，那么请给出一种方案，使得两端为同种颜色的边的数量最少（两端为不同颜色的边数量最多）。

2　问题的编码

2.1　SAT 和 MaxSAT 编码

设“顶点 i 着色为 j”这一命题的真假为 x_i^j。那么有以下约束。

首先，每个点至少着一种颜色。即：

$$\bigwedge_{i=1}^{n} \bigvee_{j=1}^{k} x_i^j \tag{1}$$

其次，每个点不能着大于一种颜色。即：

$$\bigwedge_{i=1}^{n} \bigwedge_{1\le a<b\le k} \neg x_i^a \vee \neg x_i^b \tag{2}$$

然后，相邻的点不能着同一种颜色，即：

$$\bigwedge_{u,v\in E} \bigwedge_{j=1}^{k} \neg x_u^j \vee \neg x_v^j \tag{3}$$

这三类约束都要满足，才是一种合法的染色方案。如果染色方案不一定能被满足，我们也可以将问题变成 MaxSAT，使得两端为不同种颜色的边数最多。编码方式将前两个条件看作“硬约束”，最后一个条件里面的 |E| 个用与逻辑串联起来的每一个约束看作软约束，优化目标是让这 |E| 个约束满足越多越好。后文中，为了说明方便，我们使用 $g(S)$ 来代表当变量的取值集合为 S 时，两个端点上颜色相同的边的数量。其中 $S = \{x_i^j | i = 1, 2, .., n,\ j = 1, 2, ..., k\}$. 如果将 S 方案

调整到 S’，那么 $g(S)-g(S')$ 可以表示解的优性的提升程度，这个量越大越好。在后文中，g 函数和 S 集合定义将贯穿全文。

2.2 ILP 编码

变量的定义和上一小节相同，其真假值对应了整数规划中的 0 和 1. 我们将问题编码为 0,1 整数规划问题，里面的所有变量都为 0,1 变量:

$$\max \sum_{p=1}^{m} \epsilon_p \tag{4}$$

$$s.t. \sum_{j=1}^{k} x_i^j > 0 \tag{5}$$

$$((1-x_u^j)+(1-x_v^j)) \geq \epsilon_p \qquad (u<v, \forall j=1,2,...,k) \tag{6}$$

$$(1-x_i^a)+(1-x_i^b)>0 \tag{7}$$

对于一条边，如果两个端点是同种颜色的，那么 ϵ_p 只能取 0，才能保证对于所有 j，约束（6）成立. 如果两个断电不同种颜色，j 在从 1 跑到 k 的时候，约束（6）的不等式的左边至少为 1. 所以 ϵ_p 可以给一个 1. 最终我们对于所有的边，让 ϵ_p 最大，就能够使得“两端不被染成同一种颜色”的边的数量最多。

3 图着色问题的求解

3.1 结构化搜索

对于每个连通分支，使用深度优先搜索的方法尝试顶点的染色。如果某个顶点不能被染上任意一种颜色，则进行回溯操作。在本文中，结构化搜索作为 baseline。因为该方法的运行时间有时会很长，我们限制其运行时间为 5 秒，超过 5 秒钟，算法将被强制停止，并返回“找不到解”。结构化搜索的代码实现参见 Solver.cpp 的 SystematicSearchSolver。

3.2 贪心局部搜索

我们将染色方案编码为一个长 n 的串，串里面的第 i 个位置代表顶点 i 被染的颜色。定义两个串互为近邻的含义是，两个串有且仅有一个位置不同。贪心搜索的思想是，对于当前解串 S，尝试每一个近邻 S’，求得其评估函数（这里的评估函数是 S 中两端都为同一种颜色的边的个数减去 S’ 中两端都为同一种颜色的边的个数，这个值越高，代表 S’ 的“提升效果”越好），然后选择一个评估函数值最高且大于 0 的近邻，不断迭代直到无法进行改进。贪心局部搜索的代码实现参见 Solver.cpp 的 GreedySearchSolver。在算法的具体实现时，我们观察到，当我们变换一个节点的颜色时，仅仅改变以该节点为端点的边的“约束满足与否”的回答。所以，在计算评估函数时提升量，与其我们使用整张图，分别算改变前和改变后的评估函数，不如开一个数组 count，$count[u][k]$ 记录的是以 u 为中心，邻居颜色为 k 的数量。这样，我们将 k 改为 k' 时，$count[u][k]-count[u][k']$ 就等于 $g(S)-g(S')$。这样的实现提升了算法的运行速度。不过我们不能忘记，我们每次调整完 u 的颜色后，同时更新其邻居节点的 $count$ 值。

这个方法在结束后有三种可能。第一种是找到解，第二种是搜索过程陷入了局部最优，第三种是搜索过程的最大迭代次数已到。贪心局部搜索算法的伪代码如1所示。

3.3 模拟退火方法

邻居的定义同 3.2，不同的是每一次迭代直接随机选一个近邻，如果该近邻所对应的解更优，则接受，否则以一个依赖于迭代数 t 的概率 p(t) 接受这个解。模拟退火的结束有两种可能：第一种就是找到解，第二种即最大迭代次数已到。由于我们永远有一定的概率（哪怕最后温度很低时这个概率很小）接受一个较差的解，不存在“get stuck”的情况。模拟退火的代码实现参见 Solver.cpp 的 SimulateAnnealingSolver。模拟退火的迭代次数为 100000. 我们希望在前 1 万次（10%）的迭代进行时，优化器有较高的概率（50%）接受一个非更优的邻域的取值。通过相应的可视化方法，我们确定了温度衰减参数的取值——0.9997。对于 $g(S) - g(S') = -1$ 的情况，其接受的概率随迭代步数的变化如下图所示：

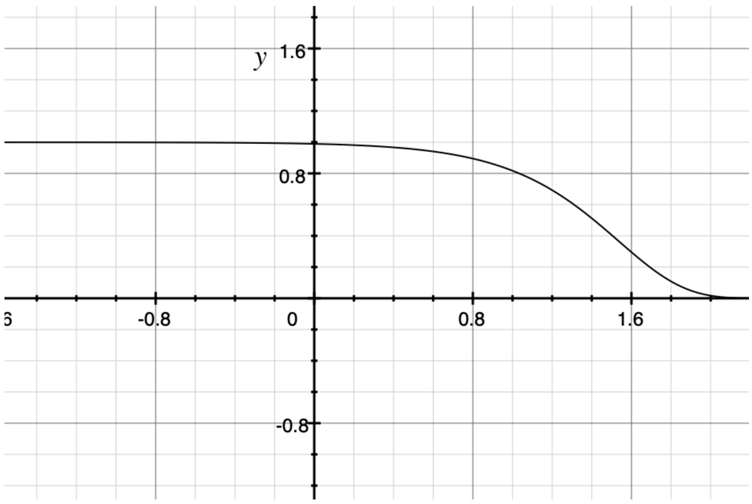

图 1: 函数 $y = exp(-\frac{1}{100*0.9997^{(}10000x-1)})$

我们固定 $g(S) - g(S') = -1$，对于温度，我们初始温度设为 100 度，之后每一次迭代按照一个比例进行衰退。经过实验，这个比例为 0.9997 比较好。于是，第 x 次迭代时，温度为 $100 * 0.9997^{(}x - 1)$。为了方便可视化，我们将横轴（迭代步数）的单位设为“万步”，于是乘以一个 10000 的系数。通过该图我们可以看出，在经过约 1.4 万步之前，程序有较高的（大于 0.5）概率可以容忍一个较差的解。这相对于我们实验指定的步数限制（10 万步），是一个比较适中的比例。

在算法的具体实现时，我们仍然借鉴了贪心局部搜索时的记录 *count* 的思想。另外，算法在优化的过程中，很可能已经找到了一个合法的染色（全局最优解）。如果直接判断 $g(S)$，时间又会很长。所以我们维护一个 not settled 的点的集合，如果该节点的颜色为 k，其任一邻居 $count[*][k]$ 都为 0，那么我们认为该点被 settle 了。settle 的集合需要随着 count 的更新而更新。

模拟退火的伪代码图2所示。

4 实验对比与结论

4.1 测试用例的生成

测试用例的生成思路如下：首先人为指定一些“母图”，母图可以自动变成相应的“子图”，也就是最终的测试用例。“母图”中的每一个节点代表子图中的一个独立集，子图中，每个独立集都由 k 个节点组成（k 可以调节）。母图中的边相当于独立集之间的连接方式，由于两个集合各有 k 个点，它们的连接共有 k^2 种可能。对于这 k^2 个中的每一种，我们都以一定的概率 p（这里是 0.8）选择“连”或者“不连”。按照上述方法生成的用例，可以保证，子图的最少染色数不超过母图。而由于母图是人为指定的，最少染色数可以很容易得出，所以当给定一个子图和母图的最少染色数作为图着色的输入时，正确的算法一定可以给出“是”的答案。

这些图的文件名形如 i_j。其中，$i = 1$，对应二分图，$i = 2$，对应至少需要三染色的图，$i = 3$，对应至少需要五染色的图。$j = 1$ 表示每个独立集有 10 个点，$j = 2$ 表示 50 个，$j = 3$ 表示 100 个。测试用例的生成代码详见 GraphMaker.cpp。

4.2 实验的设计

本文中，我们要设计实验，探索以下几个问题：

- 对于保证有解的情况，这些算法能是否够得到解？如果能够得到解，算法的速度如何？如果不能得到解，解的优性和算法的速度如何？
- 对于没有解的情况，这些算法会不会和有解的时候相比，速度有明显的差异？对于两个局部搜索算法给出的解，哪个更优？

对于算法速度的对比，我们使用该算法从开始到给出结果所占用的 CPU 时间。由于 CPU 时间是平台相关的，我们详细叙述以下实验的环境：

- Macbook Pro 2014 15.3 inch, CPU: Intel 4870hq@3.5GHz, RAM: 16GB, 1600MHz DDR3
- 编译器：Apple clang version 11.0.0，关闭编译优化 (采用 O0 选项)

实验控制的变量：

- 图的类型，这里实验了三种。分别是：最简单的二分图；母图为正方形加上一个对角线；母图为 5 个点的完全图的图
- 母图中的每个点对应的子图中的点数。即子图中的每个独立集有多少个节点。这里实验了三种，分别是 10，50，100
- 给定的颜色的数量 k。算法必须使用小于等于 k 种颜色对图进行染色。以 5 点完全图为例，如果 $k < 5$，那么会出现无解的情况。如果 $k > 5$，则会出现比 $k = 5$ 更多解的情况，算法只需给出一种染色方案即可。这里 k 实验了 2, 3, 4, 5, 6
- 搜索算法。分别是结构化搜索、贪心局部搜索与模拟退火方法。

这些受控变量形成了 3 * 3 * 5 * 3 = 135 种组合。对于每一种组合，我们分别进行了实验，并获取几项数据：算法能否得到解，算法给出的解的优性（以两端为相同颜色的边的数量来衡量，这个值越少越好），算法从开始执行到给出结果所经过的 CPU 时间。实验的进行部分请见 test.cpp。（如果需要复现代码，需要改相应的路径）

4.3 实验结果

上述 135 种受控变量的取值组合所对应的实验结果写入了结果文件 data/result.txt 中。我们对这些 raw data 进行可视化，对上一节提出的问题进行回答。

4.3.1 原问题保证有解的情况

greedy search 有时会陷入到局部最优解，对于图 2-1，250 条边和图 3-1，492 条边，其局部最优解的“两端为相同颜色的边的个数（cost function）”如下图所示。相比之下，SA 算法并没有出现在下面的表格中。也就是说，SA 在限定的迭代次数（100000）中均找到了最优解（答案解）。可见，模拟退火有助于解逃离局部最优。

```
res[(res['has_answer'] == 1) & res['cost_function'] > 0]
```

	graph	max_color	answer	has_answer	method	solved	message	cost_function	time
58	2_1	6	3	1	greedy_search	0	local_optima_reached	1	0.000286
100	3_1	5	5	1	greedy_search	0	local_optima_reached	27	0.000314
103	3_1	6	5	1	greedy_search	0	local_optima_reached	21	0.000285

图 2: 一些 greedy search 的失败情况

对于有解的情况，当我们给程序的可用染色数正好等于真实的最小染色数的时候，各个算法的性能如下图所示。可见，如果有解的时候，结构化搜索是较优的选择。贪婪搜索的表现其次，SA 方法的占用时间最长。

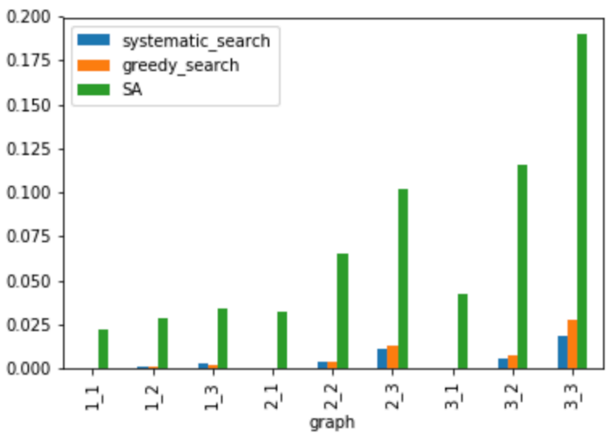

图 3: 算法时间对比图。横轴代表图的文件名，纵轴代表 CPU 时间（秒），不同颜色代表不同算法

4.3.2 原问题无解的情况

我们使用可用染色数等于真实最小染色数减一的这样的无解的情况。由于在实验中 k 从 2 开始，对于二分图来说，解都是存在的。所以下图的横轴从 2_x 开始。由于运行时间比较悬殊，我们的纵轴取了对数。

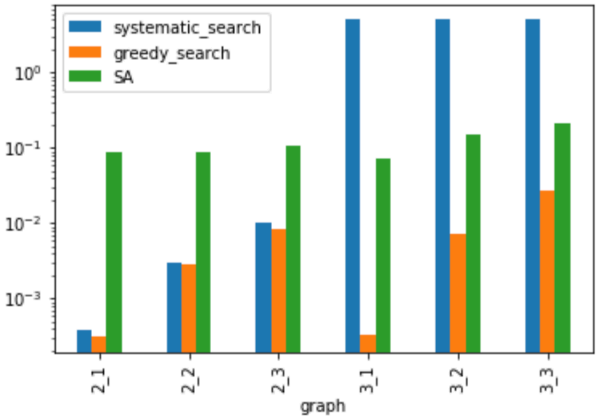

图 4: 无解情况下算法时间对比图。横轴代表图的文件名，纵轴代表 CPU 时间（秒），不同颜色代表不同算法

对于原问题无解的情况，结构化搜索的性能下降比较严重，当图的最小染色数为 5 时，程序运行超时（5 秒以上）。贪心搜索运行时间最短，但是从下图筛选出来的原因可以看出，其因为卡在了局部最优解处无法更新，所以就跳出了迭代。对于时间和 cost function 都是越小越好。

	graph	max_color	answer	has_answer	method	solved	message	cost_function	time
45	2_1	2	3	0	systematic_search	0	no_solution	250	0.000392
46	2_1	2	3	0	greedy_search	0	local_optima_reached	52	0.000313
47	2_1	2	3	0	SA	0	max_iteration_reached	52	0.087276
60	2_2	2	3	0	systematic_search	0	no_solution	6393	0.002986
61	2_2	2	3	0	greedy_search	0	local_optima_reached	1265	0.002805
62	2_2	2	3	0	SA	0	max_iteration_reached	1265	0.088617
75	2_3	2	3	0	systematic_search	0	no_solution	25173	0.010000
76	2_3	2	3	0	greedy_search	0	local_optima_reached	9507	0.008246
77	2_3	2	3	0	SA	0	max_iteration_reached	5006	0.105471
96	3_1	4	5	0	systematic_search	0	time_limit_exceed	-1	5.000000
97	3_1	4	5	0	greedy_search	0	local_optima_reached	27	0.000333
98	3_1	4	5	0	SA	0	max_iteration_reached	26	0.073433
111	3_2	4	5	0	systematic_search	0	time_limit_exceed	-1	5.000000
112	3_2	4	5	0	greedy_search	0	local_optima_reached	1072	0.007384
113	3_2	4	5	0	SA	0	max_iteration_reached	1072	0.153376
126	3_3	4	5	0	systematic_search	0	time_limit_exceed	-1	5.000000
127	3_3	4	5	0	greedy_search	0	local_optima_reached	4515	0.026818
128	3_3	4	5	0	SA	0	max_iteration_reached	4471	0.212205

图 5: 无解情况下算法时间与优性的对比数据表，由于结构化搜索没有评估函数一说，所以其结果不具有参考意义

从表中可看出，对于同一个图，SA 比贪心在有些情况下表现明显较好（尤其是图 2_3），但是其他情况下差异并不明显。由于 SA 始终有一定的概率接受较差的解（尽管后期概率很小），对于无解的情况，其程序会一直运行直到最大迭代数用完，所以执行时间也比较长。

5 总结

本文对图着色问题进行了编码，并对原问题使用了三种搜索方法，并进行了实验设计与比较。在使用局部搜索时，作者使用了一个 trick，用空间换取时间，降低了评估函数估值的时间

复杂度。在实验中，如果原问题有解，那么结构化搜索可以在比其他两个局部搜索更短的时间内准确地给出这个解。特别地，对于贪心局部搜索，有可能陷入局部最优，无法给出这个解。如果原问题无解，结构化搜索对于无解的判断是低效的。而使用局部搜索，可以在可接受的时间之内，得到一个“较优”（使得代价函数较小，即评估函数较大）解。

Algorithm 1 图染色的贪心局部搜索

Input: An initial color array $S = x_1, x_2, ..., x_n$. Graph G. mc stands for the number of colors we have.

Ouput: The result. Whether we have found the solution or max iteration reached.

```
function SOLVE(S, G, mc)
    Let count[u][k] be the number of nodes adjacent to u which has color k, initialized to zeros.
    for u ← 0 to n − 1 do
        for v ∈ Neighbor(u) do
            count[u][x_v] ← count[u][x_v] + 1
    contradict ← false
    while t < max_iteration do
        best_opt ← null
        best_value ← 0
        for u ← 0 to n − 1 do
            current_contradiction ← count[u][x_u]
            if current_contradiction > 0 then
                contradict ← true
            for k ← 0 to mc − 1 do
                if k ≠ x_u then
                    changed_contradiction ← count[u][k]
                    eval_fn ← current_contradiction − changed_contradiction
                    if eval_fn > best_value then
                        best_opt sets to "change the node u to color k"
                        best_value = eval_fn
        if contradict = false then
            Problem solved and return S.
        if best_opthasnotchanged then
            Returns that the search algorithm has stucked to local minima.
        update the count array of the neighbor of the node that we selected from best_opt
        x_{best_opt_node} ← best_opt_color
    return The solution or message or max iteration reached.
```

Algorithm 2 图染色的模拟退火

Input: An initial color array $S = x_1, x_2, ..., x_n$. Graph G. mc stands for the number of colors we have.

Ouput: The result. Whether we have found the solution or max iteration reached.

```
function Solve(S, G, mc)
    T ← 100
    decay ← 0.9997
    node_not_settled ← Φ
    Let count[u][k] be the number of nodes adjacent to u which has color k, initialized to zeros.
    for u ← 0 to n − 1 do
        for v ∈ Neighbor(u) do
            count[u][x_v] ← count[u][x_v] + 1
        if count[u][x_u] > 0 then
            node_not_settled ← node_not_settled ∪ {u}
    while t < max_iteration do
        if node_not_settled = Φ then
            Break the loop
        u ← random choose a node
        k ← random choose a color that is different from x_u ,  from 0 to mc − 1
        eval_value ← count[u][x_u] − count[u][k]
        p ← e^(eval_value/T)
        if eval_value > 0 or random events happens with p then
            for v ∈ Neighbor(u) do
                ori_v ← count[v][x_v]
                count[v][x_u] ← count[v][x_u] − 1
                count[v][k] ← count[v][k] + 1
                if ori_v > 0 and count[v][x_v] = 0 then
                    node_not_settled ← node_not_settled \ {v}
                if ori_v = 0 and count[v][x_v] > 0 then
                    node_not_settled ← node_not_settled ∪ {v}
            if ori_u > 0 and count[u][k] = 0 then
                node_not_settled ← node_not_settled \ {u}
            if ori_u = 0 and count[u][k] > 0 then
                node_not_settled ← node_not_settled ∪ {u}
            x_u ← k
        T ← decay ∗ T
    return The solution or message for max iteration reached.
```