最大团问题的分支限界算法

一、最大团问题的定义

最大团问题是指在一个简单无向图 G = (V, E)中寻找顶点最多的一个子图 C，使得∀i, j ∈ C，$(i, j) \in E$。在本文中，我们使用下列定义：

- 补图——给定一个图 G(V, E)，它的补图定义为$\overline{G}$ (V, $\overline{E}$)，其中$\overline{E} = \{(i, j) \mid (i, j) \in (V \times V) \setminus E\}$。
- 极大团——图的一个团，该团不能通过添加顶点而进一步扩大。
- 独立集——图 G = (V,E)的子图 S，其中∀i, j ∈ S，$(i, j) \notin E$。显然，图的最大团是其补图的最大独立集。
- 候选顶点——可以用来扩充当前团的顶点，即与当前团所有顶点都邻接的顶点。
- 颜色数——在着色过程中赋予顶点的自然数。
- 染色数——对图进行着色所需的最小颜色数。

分支限界算法是求解最大团问题最常用的算法。算法步骤一般分为分支与限界两部分。其中分支部分选择候选顶点集中的一个顶点，将该顶点添加到当前团中，接着进入搜索树的下一节点。在新的节点（新的递归过程）中，候选集被更新为原候选集中与刚刚被选中的顶点邻接的顶点，从而候选集中所有顶点都邻接于当前团的所有顶点。而限界部分则计算当前候选集中，最大团的规模上界，如果该上界加上当前团的顶点数不大于目前找到的最大的团，那么可以断言该分支不可能产生更好的团，因此可以裁减这一分支。

本文主要工作在于研究目前最先进的几个分支限界算法，并在这个基础上，构造出效率更好的求解算法。

二、下界与代价函数

分支限界法算法中，分支部分保证了算法的完整性与精确性，而限界部分则决定了算法的效率高低。在搜索树的某一结点上，代价函数定义为该结点可能找到的团的最大规模。为了提高算法效率，关键在于能否在合理的时间内，估算出尽可能紧的代价函数值。另外，快速找到一个大的下界，即找到一个尽可能大的团，也是提高算法效率不容忽视的部分。

估计代价函数时，一个平凡的做法是以候选集的顶点个数作为代价函数，如果当前团大小加上候选集顶点个数不超过下界，则当前分支无需进入下一轮迭代，可以进行剪枝。然而，这样的代价函数过于松散，未能精确估计团的上界，导致算法效率不高。

2.1 染色数

图的染色数可以作为最大团问题的上界（命题 1）。

命题 1 (Balas 和 Yu 1986)[1] 任意图的最大团规模不大于该图的染色数。

这个命题可以从以下事实直接得出：一个具有 k 个顶点的团至少需要用 k 种颜色进行着色，因为这些顶点都是两两邻接的，所以团中任意两个顶点不可能具有同一种颜色。

2.2 贪心着色

染色数问题所对应的判定问题，本身就属于 NP 难问题，因此使用染色数作为最大团问题的代价函数是不实际的。另一种方案则是使用贪心着色[2]的方法，求出染色数的近似解，作为最大团问题的代价函数。贪心着色过程按照给定顺序的顶点，依次对顶点进行着色，着色过程遵循如下原则：

- 如果(v1,，v2) ∈ E，则 No[v1] ≠ No[v2]
- 如果 No[v] = k > 1，则存在顶点 $v_1 \in N_R[v]$，$v_2 \in N_R[v]$，...，$v_{k-1} \in N_R[v]$，使得 $No[v_1] = 1$，$No[v_2] = 2$，....，$No[v_{k-1}] = k-1$。

其中，No[v]表示顶点 v 的颜色编号，$N_R[v]$表示顶点 v 在子图 R 中的邻居。

显然，贪心着色过程所用到的颜色数可作为最大团问题的代价函数。对于顶点个数为 n 的图，贪心着色过程的时间复杂度为 $O(n^2)$，在可接受范围之内。

2.3 下界与分支顺序

好的分支顺序能够使算法快速更新下界，从而在后续迭代中得以裁剪更多的分支。贪心着色过程为分支顺序做了很好的准备。因为团的每个顶点颜色编码都是不同的，所以在每种颜色编码的顶点中最多只能选取其中一个顶点。根据贪心着色的过程，颜色编码最大的顶点，与所有其他颜色编码的顶点集合中，恰好都能找到与之相邻的顶点，这样就使得颜色编码大的顶点属于最大团的可能比较大，所以优先选取颜色编码最大的顶点是不错的分支策略。另外，这种分支顺序使得在搜索树的同一层中，只需要进行一次贪心着色过程，因为排除了所选择的顶点后，新一轮贪心着色过程不会产生新的着色方案。

2.4 顶点顺序对着色的影响

贪心着色过程根据既定的顶点顺序，依次对它们进行着色。不同的顺序对着色效果（即所使用的颜色数）有很大的影响。直觉上，限制越多的顶点应该优先进行着色，因此，如果只考虑上界问题的话，使顶点按照度数大小降序排列，可以得到较紧的上界。这便是 MCQ[2] 算法所采用策略。

然而，正如前面所述，贪心着色过程在分支限界算法中，不仅仅是获取上界的手段，同时也对分支策略有着重大的影响。顶点被分支的次序与顶点的颜色编码大小顺序一致，且在同一种颜色编码的顶点中，越晚被着色的顶点越早被选择，因此，这里需要考虑新的顶点排序准则[3]。

将顶点排列成如下顺序V={V[1],V[2],...,V[n]}，使得在图G=(V, E)由顶点V' = {V[1], V[2], ..., V[i]} (i ≤ n) 导出的子图中，使V[i]始终都是V'中度数最小的顶点。如果{V[1], V[2], ..., V[i]}中的所有顶点都有着同样的（最小的）度数，也就是说由{V[1],V[2],...,V[i]}导出的子图是正则的，则对这些顶点进行排序时没有意义的。在这种情况下，可以直接终止该排序过程。虽然这个排序过程是比较耗时的操作，但是由于只在算法初始化的时候进行，所以从全局来看，负担是相当小的。

在MCQ算法的基础上，加上上述排序规则对顶点顺序进行初始化，便得到了MCR算法[3]。与MCQ算法相比，效率有了明显的提升。

值得注意的是，虽然在分支限界算法框架下，搜索树的同一层只需进行一次着色，然

而，在不同层上，需要对新产生的候选顶点集进行贪心着色过程。贪心着色过程对顶点顺序具有部分的继承性。当着色完成后，颜色编号相同的顶点之间仍保存着原来的顺序不变，先着色的顶点仍然在后着色的顶点之前。这样就使得下层迭代时，顶点之间是相对有序的。但是，这种顺序会在逐次迭代中失去，所以仅仅在初始阶段对顶点进行排序是不够的。考虑到排序的时间复杂度较高，必须小心权衡排序本身的效率负担与因排序所得到的效率提升。下面分别介绍两种处理策略。

2.4.1 静态顺序顶点辅助集

为了延续初始阶段顶点顺序的效果，在初始化之后，使用一个全局的顶点辅助集保存顶点的顺序。当候选顶点集需要重新着色时，可以根据顶点辅助集中的顺序，对候选顶点进行着色。由于顶点辅助集保存了初始顶点顺序之后不再更改，因此称之为静态顺序顶点辅助集。虽然静态顺序与候选顶点集所期望的顺序（即重新排序）不完全一致，但作为一种近似估计，顶点辅助集较好地平衡了排序的耗费与收益。这便是 MCS[4]所采取的策略。MCS 相比 MCR 除了增加了静态顺序顶点辅助集之外，还用到了后面即将提及的重着色技术。

2.4.2 动态排序策略

相比静态顺序一经确定便不再更改的策略，动态排序策略在特定的时刻对顶点进行重新排序，使其更加适应当时的候选顶点集。

直觉上，当子图的顶点数量较多时，重新排序的效果会越好，因为贪心着色的继承性使得排序的影响得以蔓延到更多分支。另一方面，对顶点数量较多的图，排序所花费的时间相比由于上界不紧而在无效分支上浪费的时间小得多。因此，我们应该对靠近搜索树根部的分支进行重新排序，而在离根部较远的地方放弃重新排序。

我们用 level（层）表示从搜索树根部到当前叶子节点路径上所进行的分支数（递归的次数），即当前叶子结点的深度。在寻找最大团的过程中，我们必须动态地决定在搜索树的哪一个 level 进行贪心着色前的顶点排序。例如，对于密集图而言，最大团一般比顶点数量相同的稀疏图的最大团大。我们希望在顶点数量相同的情况下，顶点排序在密集图上比稀疏图上更加频繁。另外，我们也希望在边密度一致的情况下，顶点排序在大图（顶点数量多的图）上进行的更加频繁。

我们引入了全局变量 S[level]和 S_{old}[level]，分别表示从根部到当前 level，并且包括当前 level 所进行的步数总和，以及从根部到前一 level，并且包括前一 level 所进行的步数总和，其中 S_{old}[level]是用来辅助 S[level]的计算的。我们引入变量 T[level]表示到当前 level 为止所进行的步数与当前搜索树所进行的所有步数之间的比值，即 T[level] =S[level] / ALL_STEPS，其中 ALL_STEPS 是一个全局变量，算法每进行一个分支步，其值都增加 1，我们在每个分支步中都重新计算这些值。另外，还需要一个变量 T_{limit} 来决定是否进行顶点重新排序。当 T[level] < T_{limit} 时，进行顶点重新排序，反之，当 T[level] ≥ T_{limit} 时，不进行该排序过程。

在每一 level 中，我们的算法首先更新从根部到该 level 中的步数总和 S[level] :=S[level] + S[level - 1] - Sold[level]，并且更新前一 level 的步数总和 Sold[level] := S[level - 1]。如果算法在该 level 时需要继续递归到下一 level，则将 S[level]的值加 1。关于 level 的总步数计算的例子见图 2-1。

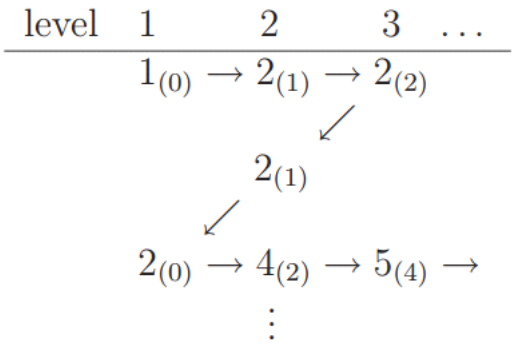

图 2-1 level 的总步数计算。列表示 level，即从根部到当前位置的递归深度。算法的搜索路径以向右的箭头表示，而向左下的箭头则表示回溯。箭头处的数值表示 S[level](S_{old} level) 。S[3] = 5 是通过如下方式计算的， S[3] = S[3] + S[2] - Sold[3] = 2 + 4 - 2 = 4，因为递归需要继续前进（向右的箭头），S[3] = S[3] + 1 = 5，Sold[3] = S[2] = 4。

其中参数 T_{limit} 的值由顶点度数的计算方式，顶点的排序准则，以及着色的策略共同决定。在 MCQdyn 算法[5]中，顶点度数使用邻接矩阵计算、顶点按照度数降序排列，而着色则采用上述贪心着色策略，通过实验证明 0.025 是 T_{limit} 比较理想的值。

由于没有辅助顶点集，所以需要使用一些技巧来增强贪心着色过程对顶点顺序的继承性。值得注意的是，贪心着色之后，在当前 level 的所有分支中，算法不会将颜色编号 No[v] < $|Q_{max}| - |Q| + 1$ 的顶点加入到当前团中，其中$|Q|$和Q_{max}分别为当前团和目前找到的最大团。为了加强着色过程对顶点顺序的继承性，MCQdyn 算法[5]在着色之后，颜色编号 No[v] < $|Q_{max}| - |Q| + 1$ 的顶点仍按照着色前的顺序排列，而其他顶点则按照颜色编号排列。这样就使得着色之后颜色编码小的所有顶点的顺序能被后续迭代完整地继承。

MCQdyn 算法是在 MCR 算法采用了动态排序策略所改进而得到的。

三、重着色

一个精确且代价小的上界，是减少不必要分支从而提升分支限界算法性能的关键。然而，减少分支数量也能通过其他方式进行，MCS[4]所用到的重着色技术便是其中一种。

重着色技术的目的在于减少不需要直接参与分支的顶点数量。根据前文所介绍的贪心着色与顶点分支方式，颜色编码小于$|Q_{max}| - |Q| + 1$ 的顶点不需要直接参与分支，因为仅由这些顶点加上当前团 Q 是不可能生成比目前最优的团更好的结果的。我们称颜色编码小于 $|Q_{max}| - |Q| + 1$ 的顶点集是被动的，而颜色编码大于等于$|Q_{max}| - |Q| + 1$ 的顶点集是主动的。重着色技术正是基于如下事实而设计的：被动顶点越多，所需要的分支数越少。重着色技术在贪心着色过程中，对主动顶点进行重新着色，使其由主动的变成被动的。但由于重着色是比较耗时的操作，因此我们在着色过程中，只对那些颜色编码等于目前颜色数量的顶点进行重着色，即 No[v] = maxno 的顶点，以平衡减少分支时间耗费与性能提升，其中 maxno 是在贪心着色过程的某一时刻，最大的颜色编码。

重着色过程如下：假设需要重着色的顶点为 v，其当前颜色编码为 k，我们首先在颜色编码为 1, 2, ..., $|Q_{max}| - |Q|$的顶点集中，即被动顶点集中，找到只有一个顶点与 v 相邻的颜色编码 k1，记这个顶点为 v1，接着我们在 k1+1，k1+2,...,$|Q_{max}| - |Q|$的顶点集中，找到可以放置顶点 v1 的颜色编码 k2，即第一个与 v1 没有相邻顶点的颜色编码，然后将 v 着色为 k1，将 v1 着色为 k2。若不存在这样的 k1 或者 k2，则无法对顶点 v 进行重着色，即保持 v

原有的颜色编码。

四、MCSMD

本文基于上述所描述的算法，提出一种改进了的算法，并将其命名为 MCSMD。该算法将 MCS 的重着色策略与静态顺序辅助顶点集、MCQdyn 的动态排序策略有机的结合在一起，并且在动态排序策略使用了与 MCR 的初始顶点排序一样的排序算法。

静态顺序辅助顶点集与动态排序策略的目的都在于优化顶点顺序对贪心着色过程的影响，然而这两种策略并非对立的。静态顺序辅助顶点集旨在加强贪心着色过程对顶点顺序的继承性，而动态排序策略则在适当的时候重新计算顶点顺序。两者的结合使得在合理的时间成本内，顶点的顺序更切合当前的需要。为了将两者结合，我们需要在每次排序之后都用一个新辅助顶点集记录顶点顺序，而不像 MCS 那样仅在初始化阶段定义辅助顶点集。

重着色是对贪心着色过程的一种改进，具有普适性，因此我们在 MCSMD 中也沿用了重着色技术。

另外，由于动态排序策略能够根据图的大小与边密度自适应地调整排序频率，使得排序在整个算法过程中的时间成本变得更加可控。这样，就使得引入效果更好同时时间成本也更大的排序规则，成为了可能。在本文提出的算法 MCSMD 中，我们在动态排序策略中使用了与 MCR 初始化顶点顺序一样的排序规则。

五、实验结果

在最大团问题的精确算法中，MCS[4]、MCQdyn[5]与 MaxSatClique[7]算法分别是目前最先进的算法之一。其中 MaxCliqueDyn 使用 MaxSAT 推理技术，它在求解代价函数时，将贪心着色后的属于同一颜色编码的所有顶点编码成 MaxSAT 问题子句，然后另外引进一些子句保证团中顶点两两相邻的属性，最后通过 MaxSAT 推理，找出其中冲突子句集的个数，进而计算出相容的颜色编码最多有多少个，其中相容颜色编码是指可由这些颜色编码的顶点共同构成一个团。

我们将本文提出的算法 MCSMD 与这三个算法进行比较。

5.1 用户时间

在最大团的准确性算法的对比当中，为了方便引用不同论文的数据，需要用一个基准程序运行若干基准测试图，从而得出所使用的计算机的“用户时间”。然后根据其他论文中的用户时间，便可以知道我们所用机器的运算速度，与其他论文作者所使用的机器运算速度之间的比值关系，从而可以将他们的实验结果规约到本文中使用。

我们所使用的是 i5 2.5GHz 的 CPU，以及 4GB 内存的计算机。在该计算机中，运行基准程序 dfmax，对 DIMACS 测试图例 r100.5，r200.5，r300.5，r400.5，r500.5，所耗费的时间分别是 0，0.02，0.19，1.19，4.48。由于对 r100.5 的运行时间是 0，无法通过比值的方式得出我们的计算机与其他机器速度之间的比例，因此我们不考虑该测试图。其他机器的运行时间，以及这些时间与我们机器所用的时间的比值如表 5-1 和表 5-2 所示。其中 MCS[4]中的用户时间有误，经由与作者沟通之后，其用户时间更正为表 5-2 中的数据。

由于 MCQdyn 作者提供了源代码，因而无需计算其用户时间，我们直接通过运行该算法

进行比较。

从表 5-1 和表 5-2 得知，我们的机器比 MaxCliqueDyn 所用的机器快 1.462 倍，比 MCS 所使用的机器快 1.928 倍。因此他们论文中的数据需要分别除于相应的倍数才能用于本文中。

表 5-1 MaxCliqueDyn 中的用户时间

Graph	r200.5	r300.5	r400.5	r500.5
MaxCliqueDyn	0.033	0.27	1.6939	6.28
Ratio with ours	1.65	1.42	1.3773	1.402
average ratio	1.462			

表 5-2 MCS 中的用户时间

Graph	r200.5	r300.5	r400.5	r500.5
MaxCliqueDyn	0.0415	0.359	2.21	8.47
Ratio with ours	2.075	1.8895	1.857	1.8906
average ratio	1.928			

5.2 随机图结果

我们首先在随机图上进行算法结果的比较。在比较过程中，不考虑顶点个数小于 200 的实例，这些实例对于这些算而言都太过简单，即都能在 1 秒之内被解决，因此没有比较的意义。而顶点数大于 500 的随机图我们也不考虑，因为测试这些实例需要耗费较长的时间，往往一个例子所占用的时间都是按照小时计算的。在我们的统计中，对于任意特定顶点个数和边密度的随机图所得到的结果，都是用 3 个随机图的平均结果而得的。MCQdyn、MCQdyn 加初始化，以及 MCSMD 在随机图上的结果如表 5-3 所示，其中 num.steps 列表示搜索树的大小，以递归次数来衡量。

从表 5-3 中可以看出，MCQdyn 的初始化操作在随机图上并没有明显的改进。将重着色的技术与动态排序策略结合起来，并在动态排序策略中改进排序准则后得到的 MCSMD 无论在递归次数还是在用耗费的 CPU 时间上，都显著地小于 MCQdyn。因此，在随机图上 MCSMD 相比 MCQdyn 以及 MCQdyn 性能更好。

表 5-3 MaxCliqueDyn、MaxCliqueDyn 加初始化、以及 MCSMD 在随机图上的比较

Graph		MCQdyn		MCQdyn + init_sort		MCSMD	
n	p	num.steps	CPU time	num.steps	CPU time	num.steps	CPU time
200	0.6	36890	0.05	30591	0.05	16340	0.04
	0.7	121188	0.23	116383	0.22	61224	0.18
	0.8	1039561	2.4	1191612	2.53	429400	1.60
	0.9	11292751	45.61	10275509	36.74	2997978	18.12
	0.95	7157365	49.64	7782767	46.16	2686360	22.13
300	0.6	293652	0.56	284600	0.55	163429	0.49
	0.7	3932576	8.07	3956420	7.86	1717364	5.83
	0.8	97714062	272.62	94837439	245.81	32564730	145.86
500	0.5	829658	1.74	803278	1.69	516816	1.60
	0.6	10637340	24.24	10684616	23.93	5642928	20.69

	0.7	289984697	791.43	310023669	806.34	138001245	605.51

规约之后 MCS 和 MaxSatClique 在随机图上的运行时间如表 5-4 所示。可以看出 MCS 几乎在所有的实例在都表现得不如 MCSMD。这表明在随机图实例上，MCSMD 更优于 MCS。

然而，在与 MaxSatClique 的比较中， MCSMD 比 MaxSatClique 好的例子有 5 个，而 MaxSatClique 比 MCSMD 好的例子有 4 个。而且彼此都有显著好于对方的例子，如 n = 200, p = 0.9 以及 n= 300, p = 0.8 的例子中，MaxSatClique 显著优于 MCSMD，而在 n = 500, p = 0.6 以及 n = 500, n = 0.7 的例子中，MCSMD 显著优于 MaxSatClique。

表 5-4 规约后 MCS 和 MaxSatClique 在随机图上的运行时间，及其与 MCSMD 的比较

Graph		MCS	MaxSatClique	MCSMD
n	p	CPU time	CPU time	CPU time
200	0.8	2.33	1.55	1.60
	0.9	38.38	6.83	18.12
	0.95	30.60	1.64	22.13
300	0.6	0.52	1.02	0.49
	0.7	6.22	7.04	5.83
	0.8	204.36	108.28	145.86
500	0.5	1.45	4.27	1.60
	0.6	20.75	37.12	20.69
	0.7	798.24	784.54	605.51

5.3 DIMACS 基准图的结果

通过实验，我们发现几乎在所有的 DIMACS 基准图实例上，本文描述的算法 MCSMD 都显著地优于 MaxCliqueDyn，因此，在本节中，我们只将 MaxCliqueDyn 与用了初始化将其改进之后的算法进行对比，并且在算法开始时对顶点进行一次贪心着色过程，做这样的对比，是为了确定初始化是否对算法带来了改进。在其他算法的比较中，我们不再考虑 MaxCliqueDyn。

表 5-5 MaxCliqueDyn 与其初始化改进之后的版本

Graph	w	MaxCliqueDyn		MaxCliqueDyn + initial	
name		num.steps	CPU time	num.steps	CPU time
brock200_2	12	3619	0.01	2702	0
brock200_4	17	48397	0.09	54195	0.09
brock400_2	29	44602709	165.75	27699632	128.55
brock400_4	33	45795157	142.35	12118105	52.55
brock800_2	24	1297531152	5309.34	971297327	4412.61
brock800_4	26	564317467	2549.93	322769789	1952.56

MANN_a27	126	49439	1.86	48075	1.78
MANN_a45	345	4647163	1194.75	4569457	1201.49
p_hat300-1	8	2159	0	1635	0
p_hat300-2	25	7698	0.02	6998	0.03
p_hat300-3	36	633573	2.41	645367	2.6
p_hat700-1	11	28060	0.06	26316	0.07
p_hat700-2	44	1104290	6.82	1257534	9.65
p_hat700-3	62	292993278	3050.37	283878927	3189.58

MaxCliqueDyn 与用初始化改进之后的算法的对比如表 5-5 所示。我们可以看到用初始化进行改进之后，所需的分支次数有了明显的减少。对于 CPU 时间来说，用初始化改进之后的算法在 brock 基准实例上性能提升了不少，而在 p_hat 基准实例则性能有所下降，在本文中，我们认为这些性能下降在可接收范围之内。

对于 MCS 与 MCSMD，我们不仅比较它们在这些实例的 CPU 时间，同时也比较它们解决这些实例所需要的递归次数，MCS 的递归次数是从 Mikhail Batsyn 等人(2013)[8]中引入了。这三个算法在递归次数上的比较如表 5-6 所示。

表 5-6 MCS 与 MCSMD 在递归次数上的比较

Graph	MCS	MCSMD
name	num.steps	num.steps
brock400_1	88555048	58983645
brock400_2	34145195	13574692
Brock400_3	66379744	5040024
brock400_4	29696341	5865953
brock800_2	972110520	505023165
brock800_4	424176492	173377989
MANN_a27	33345	23902
MANN_a45	221476	1062570
p_hat300-3	565792	205296
p_hat500-2	89836	40875
p_hat500-3	17259920	6312690
p_hat700-1	29656	17426
p_hat700-2	670369	329870
p_hat700-3	98911559	69848881
p_hat1000-2	25209207	9888902
san400_0.7_1	64568	12850
san400_0.7_2	23471	78228

san400_0.7_3	253044	260854
san400_0.9_1	20537	6536431
sanr200_0.7	115666	53848
sanr200_0.9	8103466	1924603
sanr400_0.7	51507583	20394027

如表 5-6 所示，MCSMD 在递归次数上都显著地小于 MCS，在许多实例上所需的分支数只有 MCS 的一半或更少。这说明将动态着色策略引进 MCS，并与 MCS 静态顺序的辅助顶点集结合的技巧在减少分支数上是成功。

表 5-7 规约后 MCS 和 MaxSatClique 在 DIMACS 基准图上的运行时间，及其与 MCSMD 的比较，其中在 Tomita 等人（2010）中，MCS 没有 p_hat700-1 的试验结果

Graph	MaxSatClique	MCS	MCSMD
name	CPU time	CPU time	CPU time
brock400_1	253.65	359.44	276.19
brock400_2	122.23	154.05	82.97
Brock400_3	198.40	242.74	37.11
brock400_4	114.43	331.43	39.27
brock800_2	5259.92	4340.25	4299.37
brock800_4	2653.90	2073.13	1432.27
MANN_a27	0.45	0.41	0.43
MANN_a45	174.88	145.75	116.02
p_hat300-3	1.42	1.30	1.52
p_hat500-2	0.62	0.36	0.37
p_hat500-3	38.27	77.80	67.2
p_hat700-1	0.55	——	0.09
p_hat700-2	3.33	2.90	3.89
p_hat700-3	706.57	1240.66	1086.77
p_hat1000-2	100.21	114.63	124.44
san400_0.7_1	0.14	0.28	0.1
san400_0.7_2	0.06	0.07	0.49
san400_0.7_3	0.51	0.73	1.29
san400_0.9_1	1.35	0.05	96.93
sanr200_0.7	0.26	0.18	0.17
sanr200_0.9	3.91	21.27	12.55
sanr400_0.7	96.44	93.88	74.09

MaxSatClique、MCS 以及 MCSMD 在 CPU 时间上的比较如表 5-7 所示。

考虑 MCS 与 MCSMD，在 brock、MMAN、sanr 实例上，MCSMD 整体上性能优于 MCS。而在 san 以及部分的 p_hat 实例上，MCSMD 表现得并不如 MCS，然而在这两组实例中，除了 san400_0.9_1（MCS 显著地优于 MCSMD)以及那些能够在 1 秒之内求解的实例外，这两个算法性能的差别在较小的范围之内。因此，从整体上而言，MCSMD 是优于 MCS 的。

对于 MaxSatClique 与 MCSMD，前者在 11 个实例上优于后者，而后者在 12 个实例上优于前者，而且两者都有明显优于对方的实例，例如 MaxSatClique 在 p_hat500-3、p_hat700-3 与 san400_0.9_1 等实例上显然占优，而 MCSMD 在 brock、MANN_a45 以及 sanr400_0.7 等实例上显然占优。因此，我们可以说这两个算法的性能不相上下。

六、总结与展望

分支限界算法性能的关键在于合理地计算代价函数，以及快速地取得好的下界。其中合理的估算代价函数有两方面的含义，其一为代价函数作为问题上界的精确性，其一为计算代价函数所耗费的成本与取得的收益之间的平衡。而快速地取得好的下界关键在于分支策略的优劣，一个好的分支策略使算法能够在比较短的时间内找到更优的解。另外，代价函数的计算与分支策略并非完全独立的，从本文所介绍的分支限界算法中，优先选取颜色编码值大的顶点进行分支，使得作为代价函数计算的贪心着色过程在搜索树的同一层中，只需要计算一遍即可，另一方面，贪心着色之后颜色编码值最大的顶点本身也是非常可能属于最优解的顶点。这样就使得代价函数的计算与分支策略有机地结合在了一起，极大提升了算法的效率。

一般情况下，分支限界算法的提升空间大部分在于代价函数的选择与计算。虽然下界也有改进的地方，但往往只停留在初始化阶段，例如目前有些先进的算法在初始化阶段调用随机算法，得到一个次优界作为下界，然后才进入传统的分支限界算法，从而极大地提高了算法效率。自从 2003 年[2]贪心着色被选为代价函数以来，大部分优秀地算法都集中精力于改进着色这一过程，并不断取得了突破。值得注意的是，着色固有的属性决定了它不可能作为最大团问题的精确上界。

定理 (Mycielski 1955)[9][38] 对于任意自然数 n，都存在有限个不包含三角形的图，不能用 n 种颜色进行着色。

显然，任意一个不包含三角形的图的最大团不可能大于 2。但是该图的染色数可以远大于最大团的大小。图 6-1 显示了不包含三角的图，其染色数分别为 2,3,4，它们的最大团的大小均为 2。

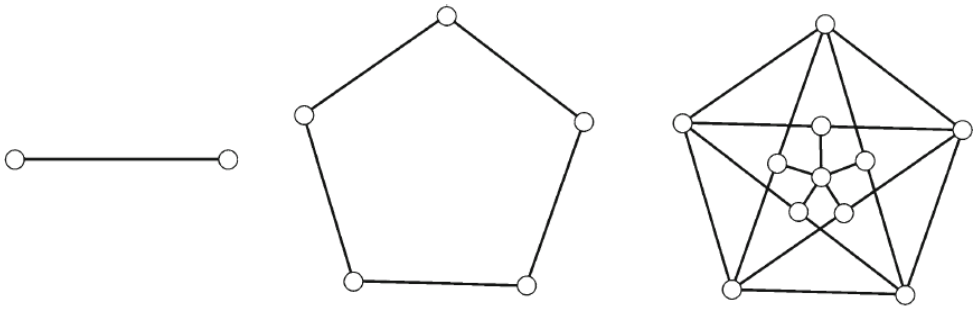

图 6-1 染色数为 2,3,4，最大团大小为 2 的 Mycielski 图

因此，除了着色之外，是否还有其他能快速计算，且更为紧的代价函数，也许是最大团问题分支限界算法的新突破口。

另一方面，将最大团问题的一部分关键步骤转换为其他问题，再使用其他问题的算法进

行计算，也是算法常用的该进途径。例如本文提到的 MaxSatClique 算法，在贪心着色之后，用 MaxSat 推理，进一步缩小上界。目前也有一些高效的算法在最大团分支限界算法的某些步骤中使用了随机算法，并取得了相当可观的性能提升。

七、参考文献

[1] Balas E, Yu CS, Finding a maximum clique in an arbitrary graph. SIAM J Comput 15(4), 1986:1054–1068.

[2] Tomita E, Seki T, An efficient branch-and-bound algorithm for finding a maximum clique. In: Proceedings of the 4th international conference on discrete mathematics andtheoretical computer science, DMTCS ' 03. Springer-Verlag, Berlin, Heidelberg, 2003:278–289.

[3] Tomita E, Sutani Y, Higashi T, Takahashi S, Wakatsuki M, A simple and faster branch-and-bound algorithm for finding a maximum clique. In: Proceedings of the 4th international conference on algorithms and computation, WALCOM'10. Springer-Verlag,Berlin, Heidelberg, 2010:191–203.

[4] Etsuji Tomita, Tatsuya Akutsu, Tsutomu Matsunaga, Efficient Algorithms for Finding Maximum and Maximal Cliques: Effective Tools for Bioinformatics in "Biomedical Engineering, Trends in Electronics, Communications and Software,"Anthony N.Laskovski (Ed.), InTech, 2011：625-640

[5] Konc, J., & Janezic, D, An improved branch and bound algorithm for the maximum clique problem. proteins, 4, 5. 2007.

[6] Li CM, Quan Z, Combining graph structure exploitation and propositional reasoning for the maximum clique problem. In: Proceedings of the 2010 22nd IEEE international conference on tools with artificial intelligence, Vol 01, ICTAI'10. IEEE, Arras, 2010a: 344–351.

[7] Li CM, Quan Z, An efficient branch-and-bound algorithm based on maxsat for the maximum clique problem. In: Proceedings of the twenty-fourth AAAI conference on artificial intelligence, AAAI-10. AAAI Press, Atlanta, 2010b:128–133.

[8] Mikhail Batsyn, Boris Goldengorin, Evgeny Maslov. Panos M. Pardalos, Improvements to MCS algorithm for the maximum clique problem. J Comb Optim DOI, 2013.

[9] Mycielski J, Sur le coloriage des graphes. Colloq Math 3, 1955:161–162.

八、附录——部分伪代码

```
procedure EXPAND(R)
begin
    while R≠ Ø do
        v := a vertex of R
        if |Q|+ UB(R) > |Q_max|          //prune
            then
            Q := Q∪{p}
            R_p := N_R(v)
            if R_p≠ Ø then
                EXPAND(R_p)
            else return
            Q := Q - {p}
        else return
        R := R - {p}
    end {of EXPAND}
```

图 8-1 分支限界算法的基本框架

```
procedure COLOR-SORT(R, C)
begin
{COLOR}
    maxno := 0;
    C[1] := Ø;
    while R ≠ Ø do
        v := the first vertex in R;
        k := 1;
        while C[k] ∩ N_R(v) ≠ Ø
            do k := k + 1 od
        if k > maxno then
            maxno := k;
            C[maxno] := Ø;
        No[p] := k;
        C[k] := C[k] ∪ {p};
        R := R - {p};
{SORT}
    i := 1;
    for k := 1 to maxno do
        for j := 1 to |C[k]| do
            R[i] := C[k][j];
            i := i + 1;
end {of NUMBER-SORT}
```

图 8-2 贪心着色与顶点分支顺序

```
procedure    SORT(R)
    begin
        i := |R|;
        R_min := set of vertices with minimum degree in R
        while |R_min|  ≠  |R| do
            v := R_min[1];
            V[i] := v;
            R := R - {v};
            i := i - 1;
            for j := 0 to |R| do
                if R[j] is adjacent to i then
                    deg_R[R[j]] := deg_R[R[j]] - 1;
                fi
            od
            R_min := set of vertices with the minimum degree in R
        od
    end {of    SORT}
```

图 8-3 顶点初始化排序

```
procedure Re-COLOR(v, R, C)
begin
    for k1 := 1 to |QMAX| - |Q| do
        if |C[k1] ∩ N_R[p]| then
            v1  = the element in C[k1] ∩ N_R[v]
            for k2 := k1 + 1 to |QMAX| - |Q| do
                if C[k2] ∩ N_R[v] = φ  then
                    {Exchange the Colors of v and v1}
                    C[No[v]] := C[No[v]] - {v};
                        C[k1] := (C[k1] - {v1}) ∪ {v};
                        C[k2] := C[k2] ∪ {v1};
                        return
    end {of Re-COLOR}
```

图 8-4 重着色过程