中国科学: 信息科学 2021 年 第 51 卷 第 4 期: 582–601
SCIENTIA SINICA Informationis
论文

求解增量二分图优化问题的动态规划驱动的局部搜索算法

彭博[1], 卢晨贝[2], 赵岳虎[2], 苏宙行[2], 廖毅[1], 吕志鹏[2*]

1. 西南财经大学工商管理学院, 成都 610074
2. 华中科技大学计算机科学与技术学院智慧计算与优化实验室, 武汉 430074
* 通信作者. E-mail: zhipeng.lv@hust.edu.cn

收稿日期: 2019–04–09; 修回日期: 2019–06–11; 接受日期: 2019–08–05; 网络出版日期: 2021–03–18

中央高校基本科研业务费专项资金 (批准号: JBK1901011, JBK190504) 和智能信息处理与实时工业系统湖北省重点实验室基金 (批准号: znxx2018QN01) 资助项目

摘要 增量二分图优化问题 (dynamic bipartite drawing problem, DBDP) 是一个具有 NP 难度的组合优化问题, 该问题在实际生产生活中有着广泛的应用. 本文提出了一种新的动态规划驱动的局部搜索 (DP-LS) 算法来求解该问题. 不同于文献中求解该问题和该类问题的所有启发式算法的邻域搜索方式 (即每次邻域操作只对一个或两个节点进行插入或交换动作), 本文提出的动态规划驱动的局部搜索算法能从邻域结构中挑选出并执行多个独立的邻域动作, 大大提高了邻域搜索的效率. DP-LS 算法从一个随机初始解出发, 迭代地利用基于动态规划的局部搜索算法来寻找局部最优解, 同时结合扰动机制跳出局部极值陷阱实现全局搜索. 本文提出的增量评估方法能够快速评估基于插入和交换的邻域动作, 可以大大提高算法的搜索效率. 本文针对 1120 个公共算例进行了计算实验并同文献中已有算法 (包括通用求解器 Gurobi) 进行对比, 表明了所提出的动态规划驱动的局部搜索算法在解的优度和计算效率两方面的有效性. 此外, 通过对比实验表明了 DP-LS 算法中动态规划机制的有效性 (提升近十倍的搜索效率). 值得注意的是, 本文提出的基于动态规划的局部搜索算法不仅能够用于求解 DBDP 问题, 也能作为一种通用的启发式算法来求解其他组合优化问题, 尤其是排序类优化问题.

关键词 增量二分图优化问题, 动态规划, 局部搜索, 增量评估机制

1 引言

基于图的布局优化理论能够通过求解图的几何表示等问题来提升其在对应领域的应用效率, 如在线广告 [1]、超大规模集成电路设计 [2]、网络管理 [3] 和软件工程 [4] 等. 尽管用于判断绘图质量的标准是相对主观的, 但是最小化图的交叉边数是一个较为公认的判别绘图质量好坏的常用标准. Erten 等 [5]

引用格式: 彭博, 卢晨贝, 赵岳虎, 等. 求解增量二分图优化问题的动态规划驱动的局部搜索算法. 中国科学: 信息科学, 2021, 51: 582–601, doi: 10.1360/SSI-2019-0122
Peng B, Lu C B, Zhao Y H, et al. A dynamic programming-based local search algorithm for solving the dynamic bipartite drawing problem (in Chinese). Sci Sin Inform, 2021, 51: 582–601, doi: 10.1360/SSI-2019-0122

在传统静态图的基础上考虑了随时间变化的实时图的动态绘制场景. 通过向原始图添加新的需求节点和边, 得到动态图 (或增量图), 因此这类图可以划分为原始部分 (原始节点、原始边) 以及增量部分 (增量节点、增量边). 在许多实际应用场景中, 用户需要保持一个所谓的 “心理图”, 当顶点和边被连续添加或者从图中删除, 用户需要调整自己的心理图以适应新图. 增量图优化问题的目标就是寻求减少这种实时动态环境下的图形差异.

在增量图优化问题中, 基于二分图的优化问题是学者们研究较多的一类问题. 对于一个二分图, 图两端的节点由直线相连, 最小化二分图中的总交叉边数问题称为二分图最小交叉问题 (BDP). 该问题已经在理论上被证明为 NP 难问题 [6], 这意味着不存在多项式时间复杂度的高效求解算法. 换言之, 对于此类难度的问题, 其计算时间一般会随着问题规模的增长而呈现出指数级别的暴增. 经典的二分图问题 BDP 在过去几十年中得到了广泛的关注. 其中最为著名的是 Eades 和 Kelly [7] 提出的基于简单节点排序的启发式算法, 以及 Jünger 和 Mutzel [8] 提出的分支分割算法. Martí 和 Etutuh [9] 提出保持原始节点的相对顺序不变, 以实现原始图和新图之间的稳定性. 此后, Martí 等 [10] 提出了增量二分图最小交叉问题, 缩写为 DBDP.

DBDP 问题主要研究如何在动态变化的图形中尽量保留原图形的布局结构, 使其具有一定的实时性和动态性, 并更加符合实际应用场景. 因此, 该问题是很多行业和领域的核心问题, 例如: 在已知的集成电路设计布图中, 如果需要增加新的半导体元器件, 元器件之间的导线交叉应当尽可能的少, 以尽量减少相互的干扰. 在社交网络中, 个体与个体之间的关系被描述成节点和边. 这些网络经常随着时间发生改变, 即使新个体增加进来, 新的网络布局应维持原布局的结构.

在过去一些年, 学者们提出了求解增量图优化问题的一系列精确算法和元启发式算法. Martí 和 Etutuh [9] 提出了一种基于分支定界的精确算法来求解 DBDP 问题, 该算法利用基于树搜索的机制来探索每层节点的排序. 此方法只能求解节点数不多于 32 个节点的小规模算例. 作者还提出了一种贪心随机自适应搜索算法 (GRASP) 来求解大规模问题算例. Martí 等 [10] 提出了一个数学规划模型, 基于 Gurobi 求解器的计算实验表明了该模型的有效性, 同时该方法能够求解 50∼100 个节点的中等规模算例. 此外, 作者还将禁忌搜索算法分别与 GRASP 以及路径重连 (path relinking) 方法相结合, 用以求解多达 471 个节点的大规模算例. 提出的禁忌搜索算法每次评估邻域结构, 并选择一个节点并将其移动到最佳位置, 同时利用一个存储结构来保存移动节点和对应位置等相关信息, 在接下来的一段迭代步数之内, 禁止该节点的移动. 除了 DBDP 问题, 该类问题的变种也得到了学者们的广泛研究. Martí 等 [11] 使用禁忌搜索和路径重连的混合算法求解增量图最小最大交叉问题. Napoletano 等 [12] 提出了求解带原始节点移动约束的 DBDP 问题. Sánchez-Oro 等 [13] 提出了一个变邻域散列搜索算法来求解增量多层图优化问题.

局部搜索算法作为一种求解 NP 难问题最有效的启发式算法之一, 其应用领域十分广泛, 包括云计算负载均衡问题 [14]、最大可满足性问题 [15], 以及节点覆盖问题 [16] 等. 针对 DBDP 问题, 本文采用动态规划驱动的局部搜索算法对其进行求解. 本文的主要贡献如下:

• 提出了一个动态规划驱动的局部搜索算法. 采用的动态规划机制能够从邻域评估中挑选出多个独立的邻域动作并同时进行执行, 提升了算法的搜索能力.

• 提出了基于两种邻域动作 (插入和交换动作) 的增量评估方法, 增强了算法的搜索效率.

• 基于 1120 个基准算例进行了测试实验, 所提出的动态规划驱动的局部搜索算法能够比已有算法以更快的收敛速度找到更好的解. 特别指出的是, 在解的优度方面, DP-LS 算法能够改进或持平 1109 个算例的最好结果; 在计算时间方面, 比 TS+PR 算法快数十倍.

• 本文提出的动态规划驱动的局部搜索算法具有一定的普适性, 可以用来求解相关的组合优化问

题, 尤其是排序类优化问题.

本文的结构组织如下: 第 2 节介绍 DBDP 的问题描述和数学模型; 第 3 节介绍求解该问题的动态规划驱动的局部搜索算法; 第 4 节介绍本文实验的运行环境和算法参数配置, 同时将 DP-LS 算法与多种算法进行详细的对比; 第 5 节对 DP-LS 算法的重要组成部分, 如邻域动作以及参数敏感性进行了分析和讨论; 第 6 节对全文进行归纳和总结.

2 DBDP 问题描述和数学模型

将 BDP 问题中的二分图定义为 $G=(V_1,V_2,E)$, 其中, V_1, V_2 和 E 分别表示左层节点、右层节点, 以及图 G 中的边. 每层节点数量分别表示为 $|V_1|=m_1$ 和 $|V_2|=m_2$. 一个绘图方案 D (即 BDP 问题的解) 由两层节点的序列 π_1 和 π_2 来表示, 标记为 $D=(\pi_1,\pi_2)$. 同时, 节点 u 和 v 在左层和右层的位置可以表示为 $\pi_1(u)$ 和 $\pi_2(v)$. 如果节点 u 在节点 v 的上方, 那么, $\pi_1(u)<\pi_1(v)$. 相似地, 如果节点 v 在节点 u 的上方, $\pi_1(u)>\pi_1(v)$.

Martí 等 [10] 在 DBP 问题的基础上进行了拓展, 提出了增量二分图最小交叉问题 (DBDP). 该问题在原始二分图 G 的基础上, 引入两个增量节点集 AV_k $(k=1,2)$ 以及两个增量边集 AE_k $(k=1,2)$ 来构成增量图. 本文将增量图表示为 $\mathrm{IG}=(\mathrm{IV}_1,\mathrm{IV}_2,\mathrm{IE})$. 其中, $\mathrm{IV}_k=V_k\cup\mathrm{AV}_k$ 表示增量图中的节点集, $\mathrm{IE}=E\cup\mathrm{AE}_k$ 表示增量图的边集. 每层节点的个数表示为 $|\mathrm{IV}_k|=n_k$ $(k=1,2)$. 本文将针对 DBDP 问题展开研究.

为了方便描述, 本文称图 G 中的节点为原始节点, 加到增量图 IG 中的节点为增量节点. DBDP 问题的目标是找到一个最佳布局使得总的交叉边数最少, 同时保持原始节点的相对顺序 (即先后续顺序关系) 不变. 不难发现, 当图中不存在原始节点时 (即原始节点集为空, 全部节点均为增量节点), BDP 问题可以看成是 DBDP 问题的一个特例. 本文用 $S=(\Pi_1,\Pi_2)$ 来表示增量图 IG 中 DBDP 问题的一个解, 其中, Π_1 和 Π_2 分别表示增量节点集 IV_1 和 IV_2 的节点排序. 节点 u 和 v 在 Π_1 和 Π_2 的排序中的位置可以表示为 $\Pi_1(u)$ 和 $\Pi_2(v)$.

为了更好地采用整数线性规划模型来描述 DBDP 问题, Jünger 和 Mutzel [8] 定义了三组 0-1 变量 c_{ijkl}, x_{ik}, 以及 y_{lj}. 其中, c_{ijkl} 表示当边 (i,j) 与 (k,l) 交叉时, c_{ijkl} 的值取为 1; 反之为 0. x_{ik} 表示如果第 1 层节点 i 在第 1 层节点 k 的上方, x_{ik} 取值为 1, 否则取值为 0. y_{lj} 表示如果第 2 层节点 l 在第 2 层节点 j 的上方, y_{lj} 取值为 1, 否则取值为 0. 下面给出 DBDP 问题的数学模型:

$$\mathrm{Min}\sum_{(i,j),(k,l)\in\mathrm{IE}}c_{ijkl} \tag{1}$$

$$\text{s.t. } x_{ik}+y_{lj}-c_{ijkl}\leqslant 1;\ \ \forall(i,j),(k,l)\in\mathrm{IE},\ i<k,\ j\neq l, \tag{2}$$

$$x_{ki}+y_{jl}-c_{ijkl}\leqslant 1;\ \ \forall(i,j),(k,l)\in\mathrm{IE},\ i<k,\ j\neq l, \tag{3}$$

$$x_{ij}+x_{jk}+x_{ki}\leqslant 2;\ \ \forall i,j,k\in\mathrm{IV}_1,\ i<j,\ i<k,\ j\neq k, \tag{4}$$

$$y_{ij}+y_{jk}+y_{ki}\leqslant 2;\ \ \forall i,j,k\in\mathrm{IV}_2,\ i<j,\ i<k,\ j\neq k, \tag{5}$$

$$x_{ik}+x_{ki}=1;\ \ \forall 1\leqslant i<k\leqslant n_1, \tag{6}$$

$$y_{jl}+y_{lj}=1;\ \ \forall 1\leqslant j<l\leqslant n_2, \tag{7}$$

$$x_{ij}=1;\ \ \forall i,j\in V_1:\Pi_1(i)<\Pi_1(j), \tag{8}$$

$$y_{ij}=1;\ \ \forall i,j\in V_2:\Pi_2(i)<\Pi_2(j), \tag{9}$$

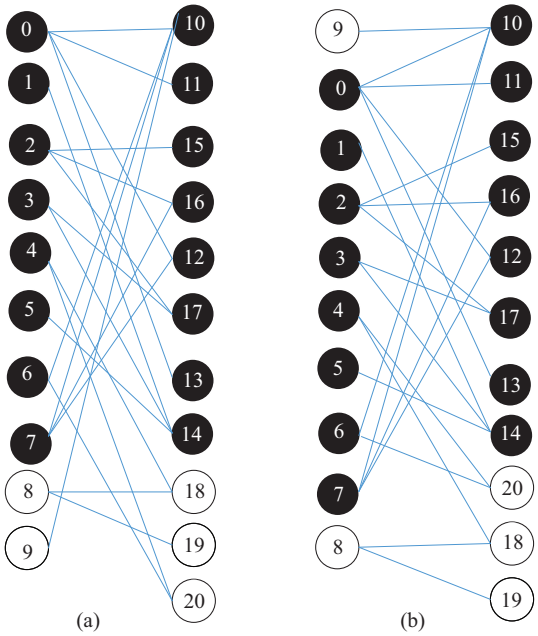

图 1 (网络版彩图) DBDP 问题示例 (GB_1_rnd1_01_0001_20 算例)

Figure 1 (Color online) An example (GB_1_rnd1_01_0001_20 instance) of two solutions for a two-layered graph. (a) the first solution (64 crossing edges); (b) the second solution (44 crossing edges)

$$x_{ij}, y_{ij}, c_{ijkl} \in \{0,1\}; \quad \forall (i,j),(k,l) \in \mathrm{IE}, \tag{10}$$

其中, 约束 (2) 和 (3) 表示如果边 (i, j) 和 (k, l) 交叉, 则变量 c_{ijkl} 取值为 1. 约束 (4) 和 (5) 是一个三重循环不等式, 加上约束 (6) 和 (7), 保证了解为一个序列. 约束 (8) 和 (9) 限定了原始节点的相对顺序不变. 约束 (10) 限定了决策变量为 0-1 变量.

增量绘图优化问题是图形可视化中的一个重要问题. 文献 [1∼4] 中介绍了多个关于该问题及其相关问题的实际应用. 在关系网络中, 成员和团体通常被描述为节点, 边则代表团体和成员之间的关系. 在实际应用场景中, 这种关系网络经常会随着时间的变化而发生改变, 这往往意味着新的团体和成员会被添加进来. 在这种实时更新的场景中, 新布局往往需要既美观又保持一定的动态稳定性 (即能够很好地满足节点的顺序关系). 图 1 展示了一个由 21 个节点构成的二分图问题的示例. 其中, 黑色节点代表原始节点, 应该保持其相对位置不变, 而白色节点代表增量节点. 图 1 中第 1 种方案存在 64 个交叉边, 能够通过 DBDP 问题的求解 (即重新分配新的白色节点的位置, 最小化总的交叉边数同时保持原始节点的先后续顺序关系不变), 将第 1 个格局优化为第 2 个格局的 44 个交叉边 (即将节点 9 放到节点 0 的上方, 将节点 20 插入到节点 14 和 18 的中间, 同时保证第 1 层的节点 0∼7 的先后续顺序关系不变, 第 2 层的节点 10∼17 的先后续顺序关系不变).

3 动态规划驱动的局部搜索算法

3.1 算法的基本原理和整体框架

先前研究该问题和该类问题的所有启发式算法在邻域搜索时均只对一个或两个节点进行邻域操作, 而本文提出的 DP-LS 算法将动态规划与局部搜索算法相结合, 能够利用动态规划机制挑选出邻域

中相互独立的一组邻域动作并同时执行. DP-LS 算法的整体框架如算法 1 所示. DP-LS 算法首先产生初始解 S, 然后利用 dpls 主搜索算法, 选择当前解中多个相互独立的邻域动作, 将当前解改进为局部最优解 S^*, 然后利用扰动算符 Perturbation 对历史最优解 S^{best} 进行扰动操作, 使其跳出局部最优值陷阱. 整个算法的搜索过程迭代执行, 直至历史最优解的未改进迭代次数 θ 达到最大阈值 Θ 为止. 历史最优解保存在 S^{best} 中, 作为算法的输出. 下面将详细介绍 DP-LS 算法的各个组成部分.

Algorithm 1 The main framework of the dynamic programming based local search (DP-LS)

Input: $G = (\text{IV}, \text{IE})$;
Output: The best found solution S^{best}.
Randomly generate an initial solution S;
$\theta \leftarrow 0$; /*The number of consecutive non-improving local optima*/
while the maximum number of consecutive non-improving global optima is not reached ($\theta < \Theta$) **do**
  $S^* \leftarrow \text{dpls}(S)$;
  if the optimum solution S^* is better than the optimum solution S^{best} **then**
    $S^{\text{best}} \leftarrow S^*$;
    $\theta \leftarrow 0$;
  else
    $\theta \leftarrow \theta + 1$;
  end if
  $S \leftarrow \text{Perturbation}(S^{\text{best}})$;
end while
return S^{best}.

3.2 搜索空间和初始化

DBDP 问题的目标是找出二分图中节点的合理排序并使所有边的总交叉数最少. 由于原始节点的相对顺序不变, 所以该问题只需要给出增量节点的排列位置. 假定第 1 层原始节点和总节点的个数分别为 m_1 和 n_1, 对于第 1 层增量节点的所有可能的排序方式数量则为 $A_{n_1}^{n_1-m_1}$; 同理, 对于第 2 层增量节点, 所有可能的排序方式数量为 $A_{n_2}^{n_2-m_2}$. 因此, DBDP 问题的搜索空间可以表示为 $A_{n_1}^{n_1-m_1} \times A_{n_2}^{n_2-m_2}$. 考虑到该问题庞大的搜索空间, 本文所采用的 DP-LS 算法只考虑在可行解空间进行搜索.

通常来说, 当主搜索算法的搜索性能足够强大时, 初始解的优劣对算法整体性能的影响不是很大 [17]. 本文采用随机算法来构造 DBDP 问题的初始解. 增量图优化问题的节点分为两种类型, 一类是原始节点, 该类节点需要保持节点的相对先后顺序不变; 另一类是增量节点, 此类节点没有约束限制, 可以任意放置. DP-LS 算法的初始解生成方式为: 先固定原始节点的位置, 然后迭代地将增量节点进行随机放置. 由于原始节点并未移动, 其相对位置是保持不变的, 因此通过此方法产生的解一定是合法解.

3.3 基于动态规划的局部搜索算法

在邻域搜索过程中, 通常需要对当前解构造邻域结构. 针对 DBDP 问题, 本文采用两种较常用的邻域动作 (插入和交换) 来构造邻域结构. 其中, 插入动作定义为将一个增量节点移动到另一个节点的上方或者下方, 而交换动作定义为将两个不同的增量节点交换位置. 值得注意的是, 两种邻域动作均只能针对相同层的节点进行操作. 当节点 u 在 v 上方时, 将节点 u 插到节点 v 下方的邻域动作记作 $\text{insert}(u, v)$; 将节点 v 插到节点 u 的上方的邻域动作记为 $\text{insert}(v, u)$. 将两个节点 u 和 v 交换位置的邻域动作记作 $\text{swap}(u, v)$.

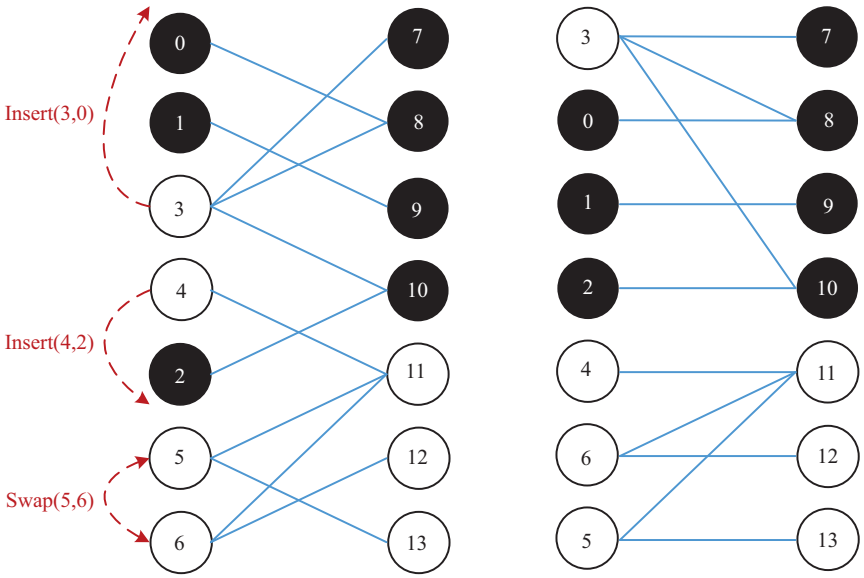

图 2 (网络版彩图) 动态规划驱动的局部搜索算法选取相互独立邻域动作的示例

Figure 2 (Color online) An example of the DP-LS method for selecting several independent neighborhood moves

通过对文献中提出的算法 (如 GRASP 算法 [9,11]、禁忌搜索算法 TS [10]、路径重连算法 PR [12]、变邻域搜索算法 VNS [13] 等) 的研究发现, 这些方法所采用的邻域搜索在邻域评估时, 仅对一个或两个节点进行插入或交换的邻域动作. 而通过对 DBDP 问题结构的分析, 不难发现在大多数情况下, 邻域结构中存在多个邻域动作相互独立的情形, 特别是基于增量图中距离较远的节点的邻域动作, 两者之间往往不会相互影响. 也就是说, 当执行一个动作之后, 另一个动作对目标函数的改变量没有发生任何变化.

为了更好地描述提出的 DP-LS 算法, 本文对邻域动作的独立性给出如下定义. 给定两个邻域动作分别为 $\mathrm{Move}_1(u_1, v_1)$ 和 $\mathrm{Move}_2(u_2, v_2)$, 如果节点 u_1 到 v_1 之间的所有节点 (包括节点 u_1 和 v_1) 与节点 u_2 到 v_2 之间的所有节点 (包括节点 u_2 和 v_2) 不存在交集, 那么, 称邻域动作 Move_1 和 Move_2 是相互独立的. 而对于一个邻域动作而言, 任何一个与之独立的邻域动作的执行, 均不会改变其对目标函数值的改进量. 因为两者之间没有公共的节点交集, 记录节点对交叉边数的数据结构的值均保持不变.

图 2 给出了一个二分图中相互独立的邻域动作的示例. 对于第 1 层增量节点 (3, 4, 5, 6), 3 个邻域动作 insert$(3, 0)$, insert$(4, 2)$, 以及 swap$(5, 6)$ 是相互独立的, 因为任何一对节点之间均不存在交集. 这就意味着, 其中任何一个邻域动作的执行, 均不会改变其他两个邻域动作对目标函数值的改进量. 对于邻域动作 insert$(3, 0)$ 而言, 其对目标函数值的改进量为 1. 如图 2 所示, 在左图节点 0, 1 和 3 对应的边中, 边 (3, 7) 与边 (0, 8) 和 (1, 9) 交叉, 同时边 (3, 8) 与边 (1, 9) 交叉, 所以当前的交叉数为 3. 而右图中仅有边 (3, 10) 与边 (0, 8) 和 (1, 9) 交叉, 此时节点 0, 1 和 3 对应的边的交叉数仅为 2. 因此基于 insert$(3, 0)$ 动作的目标函数值的改进量为 1. 此外, 可以看出先执行 insert$(4, 2)$ 或者 swap$(5, 6)$, 或者同时执行, 均不会影响 insert$(3, 0)$ 对目标函数值的改进量. 基于这一性质, 本文提出了一种动态规划机制驱动的局部搜索算法, 能够从邻域结构中同时挑选出多个相互独立的邻域动作并同时执行, 以提升邻域搜索效率.

对于增量图中任意一层 (不失一般性地假定为第 k 层), 本文用 $\delta_k(i,j)$ 来表示基于第 k 层第 i 和 j 个节点 (即节点 u 和 v), 采用插入或交换等邻域动作所带来的目标函数值的最大减少量. 式 (11) 给出了 $\delta_k(i,j)$ 的形式化定义:

$$\begin{aligned}&\delta_k(i,j)=\max\{0,-\Delta_{\text{insert}}(v,u),-\Delta_{\text{insert}}(u,v),-\Delta_{\text{swap}}(v,u)\},\\&u=\omega(i),\ v=\omega(j),\ 1\leqslant i\leqslant j\leqslant n_k,\ 1\leqslant k\leqslant K,\end{aligned}\tag{11}$$

其中, $\Delta_{\text{insert}}(v,u)$, $\Delta_{\text{insert}}(u,v)$, $\Delta_{\text{swap}}(v,u)$ 分别表示 3 种邻域动作 $\text{insert}(u,v)$, $\text{insert}(v,u)$ 和 $\text{swap}(v,u)$ 所带来的目标函数的改进量. 其中, u 和 v 分别代表第 i 和 j 个位置所对应的节点, 表示为 $u=\omega(i)$ 和 $v=\omega(j)$.

同时定义 $\text{dp}_k(i)$ 表示第 k 层第 1 个节点至第 i 个节点之间所有节点采用多个独立动作所带来的目标函数值的最大改进量. 利用式 (12) 定义的状态转移方程, 能够快速计算出 $\text{dp}_k(i)$ 的值以及其所对应的邻域动作组合:

$$\text{dp}_k(i)=\begin{cases}0, & \text{if } i=1,\\ \delta_k(1,2), & \text{if } i=2,\\ \max\limits_{1\leqslant q\leqslant i-1}\{\text{dp}_k(q)+\delta_k(q+1,i)\}, & \text{otherwise.}\end{cases}\tag{12}$$

Algorithm 2 The main search procedure of the DPLS algorithm dpls(S)

Input: The current solution S;
Output: The local optimal solution S^*.
$S^* \leftarrow S$;
$k \leftarrow 1$; /*The layer to be evaluated*/
noimprove_layer $\leftarrow 0$; /*The layer without better neighborhood solutions*/
while the termination condition (i.e., There is no better neighborhood solution for both the first and second layers: noimprove_layer $\geqslant 2$) is not reached **do**
  According to Eq. (12), calculate the maximum improvement value $\text{dp}_k(n_k)$ corresponding to the independent neighborhood move combination (Without loss of generality, assume that the optimal neighborhood move combination contains q neighborhood moves, i.e., $\text{Move}_1, \text{Move}_2, \ldots, \text{Move}_q$);
  if $q=0$ **then**
    noimprove_layer $\leftarrow$ noimprove_layer $+ 1$; /*There is no better neighborhood solution in the current layer.*/
  else
    $S \leftarrow S \oplus \text{Move}_1 \oplus \text{Move}_2 \oplus \cdots \oplus \text{Move}_q$; /*If there exists a better neighborhood solution in the current layer, then the optimal neighborhood move combination is performed.*/
    $S^* \leftarrow S$;
    noimprove_layer $\leftarrow 0$;
  end if
  if $k=1$ **then**
    $k \leftarrow 2$;
  else
    $k \leftarrow 1$;
  end if
end while
return S^*.

基于动态规划的局部搜索算法如算法 2 所示. DP-LS 算法首先初始化待评估的层数 k 为 1 (即表示从第 1 层开始评估) 以及不存在更优的邻域解的层数 noimprove_layer 为 0 (即表示左右两层都存

$$M_1=\begin{bmatrix} - & 0 & - & - & 0\\ 1 & - & 0 & 0 & 0\\ - & 2 & - & - & 2\\ - & 1 & - & - & 1\\ 1 & 0 & 0 & 0 & - \end{bmatrix} \quad M_2=\begin{bmatrix} - & 0 & - & 0\\ 2 & - & 1 & 2\\ - & 1 & - & 1\\ 2 & 2 & 1 & - \end{bmatrix}$$

图 3 记录两点之间交叉边数的矩阵 M_1 和 M_2

Figure 3 Matrices M_1 and M_2 for recording the numbers of crossing edges of pairs of vertices

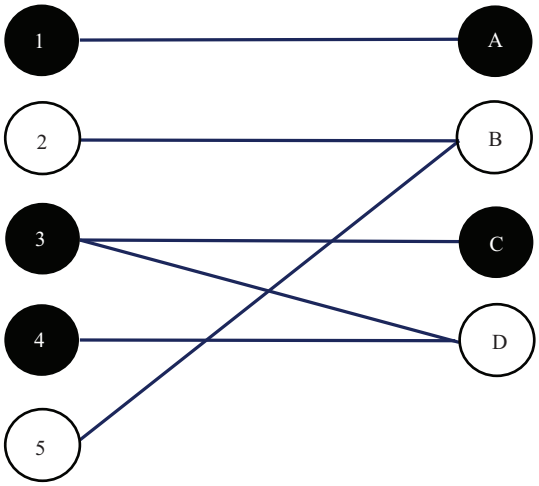

图 4 一个拥有 9 个节点的二分图示例

Figure 4 An example of a two-layered graph with nine vertices

在可能改进当前解的邻域解). 然后, 根据式 (12) 对第 k 层节点构造的邻域结构进行评估并从中挑选出相互独立的邻域动作 (假设 q 为邻域动作个数). 如果当前层不存在更优的邻域解 (即 q 值为 0), 则 noimprove_layer 值加 1, 表示当前层中不存在更优的邻域解. 否则执行最优的邻域动作组合并初始化 noimprove_layer 为 0. 随后, 将 k 变换为 2 或 1, 开始评估另一层节点的邻域动作组合. 整个算法过程迭代进行, 直至两层节点的邻域内均不存在更好的邻域解为止, 最后将 S^* 作为局部最优解返回. 假定每层的平均节点数为 $\bar{n}$, 则 DP-LS 算法主搜索过程每一次迭代的时间复杂度为 $O(\bar{n}^2)$.

3.4 增量评估机制

DP-LS 算法在进行邻域搜索时, 往往存在着大量邻域动作需要评估. 因此, 如何高效地评估不同节点之间的插入和交换动作是影响算法搜索性能的重要环节. 事实上, 当每次执行完一次邻域动作后, 对于图布局中的大多数节点对而言, 其对应的交叉边数是保持不变的. 因此, 可以利用数据结构将某个节点对的交叉数等相关信息保存, 以避免重新计算每个节点对的交叉边数, 从而更高效地计算出整个图布局的总交叉边数.

本文采用了一种基于二维矩阵的机制来记录两个节点之间的交叉边数, 能够非常高效地评估候选邻域动作的目标函数值的变化量. 给定增量图 IG 中两个节点 u 和 v, 节点 u 和 v 属于同一层. 用 NC_{uv} 表示当节点 u 处在 v 上方时, 与 u 相连的边和与 v 相连的边的交叉数; 当节点 v 在 u 上方时, 则表示为 NC_{vu}. 那么, 第 k 层 (即第 1 层或第 2 层) 的矩阵 M_k 可以表示为:

$$M_k(u,v)=[\mathrm{NC}_{uv}]_k,\quad \forall u\in \mathrm{IV}_k,\ v\in \mathrm{IV}_k,\ u\neq v,\ k\in\{1,2\}. \tag{13}$$

图 3 通过一个例子来说明如何用两个矩阵来记录图 4 中任一对节点的交叉边数. 如图 3 所示, 与节点 3 相连的边 (即 (3, C) 和 (3, D)) 同与节点 5 相连的边 (即 (5, B)) 的交叉边数为 2, 即边 (3, C) 和 (3, D) 均与 (5, B) 交叉. 因此, $\mathrm{NC}_{35}=2$. 另一方面, 如果将节点 5 插入到节点 3 的上方, 那么边 (3, C) 和 (3, D) 将不会与 (5, B) 交叉. 因此, 图 3 中矩阵 M_1 的第 5 行第 3 列的值为 0 (即 $\mathrm{NC}_{53}=0$). 类似地, 由于与 A 相连的边 (即 (1, A)) 没有和与 B 相连的边 ((2, B) 和 (5, B)) 交叉, $\mathrm{NC}_{\mathrm{AB}}=0$. 反

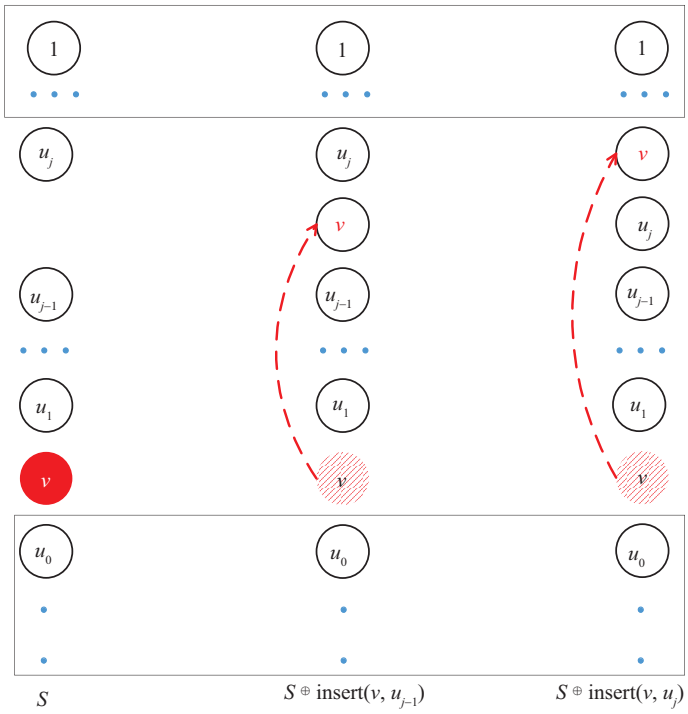

图 5　(网络版彩图) 将节点 v 插入到节点 u_{j-1} 或 u_j 之前

Figure 5　(Color online) Insert vertex v immediately before (upwards) vertex u_{j-1} or u_j

之, 若节点 B 移动到节点 A 的上方, 与 A 相连的边 (1, A) 将同两条与 B 相连的边 (即 (2, B) 和 (5, B)) 交叉, 则 $\text{NC}_{\text{BA}} = 2$. 值得注意的是, 两个原始节点由于无法同时移动, 对应矩阵中的值用符号 "–" 来替代.

基于上述两个记录节点对交叉边数的矩阵, 本文提出了一种快速评估机制计算每个邻域动作的目标函数值. 本文将通过一个示例来描述整个邻域评估过程. 当节点 v 插入到节点 u_{j-1} 或 u_j 的上方时, 如图 5 所示. 对于两个邻域动作 (即 $\text{insert}(v, u_{j-1})$ 与 $\text{insert}(v, u_j)$), 图 5 中两个方形框所标记的节点的位置未发生变化, 意味着这些节点所对应的交叉边数在邻域动作前后是相同的. 因此, 只需要考虑节点 u_{j-1} 和 v, 以及它们中间的节点的交叉数的变化值.

给定当前解 S 以及由邻域动作 $\text{insert}(v, u_{j-1})$ 产生的邻居解 $S \oplus \text{insert}(v, u_{j-1})$, 那么, 由邻域动作 $\text{insert}(v, u_j)$ 产生的邻居解的目标函数值可以由式 (14) 快速计算:

$$f(S \oplus \text{insert}(v, u_j)) = f(S \oplus \text{insert}(v, u_{j-1})) - M(u_j, v) + M(v, u_j), \tag{14}$$

其中, M 代表节点 v 所在层的矩阵, 节点 v, u_{j-1} 和 u_j 属于同一层, 并且节点 u_{j-1} 位于节点 v 和 u_j 之间.

为了高效地评估所有的邻域解, DP-LS 算法首先计算第 1 个邻居解 (即将节点 v 放到其上方的节点 u_1 的上方得到的邻居解) 的目标函数, 其目标函数值的计算公式如下所示:

$$f(S \oplus \text{insert}(v, u_1)) = f(S) - M(u_1, v) + M(v, u_1), \tag{15}$$

然后, 迭代地利用式 (14), 快速地计算出将节点 v 插入到其他位置的目标函数值的变化量. 同样地, 将节点 v 向下移动到其他节点后, 其目标函数的评估能够利用类似的方式迭代计算. 由于 DP-LS 算法只

针对增量节点进行邻域操作, 因此产生的邻居解一定是合法解.

当通过以上方法快速地评估得到所有邻居解之后, 对于基于不同节点的插入动作 (insert) 的目标函数值的改变量也可以由式 (16) 计算得到:

$$\Delta_{\text{insert}}(v,u) = f(S \oplus \text{insert}(v,u)) - f(S). \tag{16}$$

对于交换动作 (swap), 不难看出, 任意两个节点的交换动作实际上等价于两次插入动作. 比如: 节点 v 和 u 交换的邻域动作 swap(v,u) (不失一般性地假定节点 u 在 v 的上方, 同时节点 z 为节点 v 上方的相邻节点), 可以看成先将节点 v 插入到节点 u 的上方 insert(v,u), 然后再将节点 u 插入到节点 z 的下方 insert(u,z). 同样地, 为了保证产生的邻居解满足原始节点的相对顺序约束, 在交换动作中, 同样只考虑两个增量节点交换. 任意两个节点的交换动作所带来的目标函数值的变化量可以由式 (17) 计算得出:

$$\Delta_{\text{swap}}(v,u) = \Delta_{\text{insert}}(v,u) + \Delta_{\text{insert}}(u,z), \tag{17}$$

其中, 节点 v, u 和 z 属于同一层, 并且节点 z 位于节点 v 的正上方.

因此, 通过上述快速评估方法能快速地评估由插入和交换动作得到的邻居解的目标函数值. 不难发现, 第 k 层邻域评估过程的时间复杂度为 $O(n_k^2)$.

3.5 扰动算符

DP-LS 算法由于可以同时对多个独立的邻域动作进行操作, 因此是一个集中性很强的算法, 具有很快的收敛速度. 而为了避免算法陷入局部最优解陷阱, 更好地平衡搜索的集中性和疏散性, 在 DP-LS 算法中, 采用一种扰动算符对历史最优解进行扰动, 以跳出局部最优值陷阱. 具体地, 当 DP-LS 搜索算法陷入到局部最优解陷阱时 (即找不到任何能够改进当前解的邻域动作时), DP-LS 算法先随机从图中删除 α 比例的增量节点, 然后再将删掉的节点随机地重新放回图中. 通过这种扰动方式, 使得 DP-LS 算法能够跳出局部最优值陷阱, 使算法能到新的搜索区域进行搜索. 扰动算符作为一种随机疏散策略, 通常会使解变差, 以逃离局部最优值陷阱, 这有别于只会使目标函数值单调下降的局部搜索过程. 因为扰动过程是对 α 比例的增量节点进行重新放置, 考虑最坏情况即如果所有的 n 个节点均为增量节点, 则 $\alpha \times n$ 个增量节点都需要重新放置, 因此其时间复杂度为 $O(n)$.

4 计算结果

4.1 基准算例与实验配置

与文献 [9, 10] 相同, 本文采用了两组公共的基准算例集来进行计算实验. 第 1 组算例集共包含 120 个算例, 由 Martí 和 Estruch 提出 [9]. 该组算例中每层原始节点的数量 (n_1, n_2) 介于 [25, 50] 之间, 图的密度 d 介于 [0.065, 0.300] 之间. 此外, 增量图的节点数和边数与原始图的节点数和边数保持一定的比率关系, 即 ($|\text{IV}_i| = \gamma|V_i|$ 和 $|\text{IE}_i| = \gamma|E_i|$). 第 2 组算例集包含 1000 个算例, 由 Stallmann 等 [18] 提出的算例生成器生成. 该组算例第 1 层节点的数量在 [10, 377] 之间, 第 2 层节点的数量在 [10, 190] 之间. 边数的取值范围为 20~950, γ 值的大小设定为 1.1, 1.2 和 1.3.

本文提出的 DP-LS 算法用 C++ 语言编程实现, 程序运行在主频为 2.60 GHz, 酷睿 i7 处理器以及 win10 操作系统环境下. 为了更好地评估 DP-LS 的算法性能, 本文选取文献中最好的几种启发式算法作为对比的参考算法, 这些算法均运行在 2.8 GHz 的因特尔酷睿 i7 处理器以及 16 GB 的 RAM 环境下, 这些算法包括:

- 随机贪心自适应搜索算法 (GRASP) [9];
- 禁忌搜索和路径重连算法 (TS+PR) [10];
- 通用求解器 Gurobi.

为了进行公平的比较, 本文采用了一个基于 CPU 速度和 CPU 主频线性近似相关的假设. 观察到 Martí 等 [10] 所使用的 CPU 速度略快于本文使用的 CPU 速度. 因此, 直接进行计算速度的比较方式是公平的. 此外, 为了与文献 [9,10] 中报道的结果保持一致的测试强度, 限定 DP-LS 算法只运行一次. 本文的测试所用的 1120 个基准算例以及 DP-LS 算法的源代码均公开共享在网站中[1)].

算法的参数通常会对算法的性能产生一定的影响. 本文通过计算实验 (包括 5.2 小节的参数敏感性分析) 确定了 DP-LS 算法中两个参数的配置:

- 算法终止参数 Θ. 表示历史最优解在多少迭代步数内无法改进则算法终止. 在本实验中, 该参数配置为 1000.
- 扰动参数 α. 表示扰动中删除增量节点的比率 (删除节点个数为 $\alpha \times |\mathrm{AV}|$). 在本实验中, 该参数配置为 0.5.

4.2 代表性算例的计算结果

文献 [9,10] 从所有 1120 个算例中选取了 22 个算例作为训练算例集, 并报道了这 22 个算例的详细计算结果. 本文首先采用相同的方式在表 1 中报道了这部分算例的计算结果. 如表 1 所示, 第 1 和 2 列表示每个算例的两层节点个数, 第 3 和 4 列表示算例的密度和节点的比率关系 γ, 第 5 列报道了此前所有算法的最好结果, 第 6~17 列分别列出了 GRASP 算法、TS+PR 算法、Gurobi 求解器, 以及本文提出的 DP-LS 算法所找到的总交叉边数 Cross, 程序运行时间 Time (s) 以及与最好解之间的百分比差 GAP (%). #Avg 行给出了基于 22 个测试算例在上述评价指标上的平均值. #Best 行给出了各个算法所能找到的最好解的算例个数. 表中黑体数值表示最优结果. 通过观察可以发现, 本文提出的 DP-LS 算法能够找到 22 个测试算例中 21 个算例的最好结果, 同时在平均计算时间方面, DP-LS 算法仅仅需要 0.59 s, 比通用求解器 Gurobi 的 1102.40 s 快接近 2000 倍, 比 GRASP 算法的 308.08 s 快 600 多倍, 比当前文献中最好的算法 TS+PR 的 41.90 s 快了接近 100 倍. 最后一行给出了 DP-LS 算法和其他对比算法基于非参数 Friedman 测试的计算结果. 所有的 p-value 都小于 0.01, 表明比较的参考算法和 DP-LS 算法之间存在显著性差异. 通过本小节基于测试算例的比较实验, 表明了 DP-LS 算法在解的优度和计算效率上要优于文献中最好的 3 种算法.

为了验证在更大规模的算例上的算法性能, 本文选取了 9 个大规模算例作为测试算例. 其中, 每个算例的总节点数均为 471. 同时, 本文还选取了文献中最好的启发式算法 TS+PR 以及通用求解器 Gurobi 作为对比的参考算法, 实验结果如表 2 所示. DP-LS 算法能够在 9 个大规模算例上, 以绝对的速度优势找到更好的解 (9.63 s vs. 598.16 s 以及 38500.89 vs. 41465), 验证了 DP-LS 算法在大规模算例上的有效性和高效性.

4.3 第 1 组算例集的计算结果

本小节针对第 1 组基准算例集, 选取了 4 种算法 (GRASP, TS(500)+PR, TS+PR 和 Gurobi) 进行结果对比. 其中, TS(500)+PR 算法是 TS+PR 算法的一个变种算法. 表 3 报告了所有对比算法在第 1 组算例集 (共 120 个算例) 上的计算结果. 可以看出, 对于总结点数分别为 50, 75 和 100 的不同规模的算例, DP-LS 算法在一组算例中的平均交叉边数 ACross、平均计算时间 ATime (s), 以及与最好解之

1) https://github.com/283224262/graph-drawing.

表 1　DP-LS 算法和 3 个参考算法 (GRASP, TS+PR 和 Gurobi) 基于 22 个中小规模算例的实验结果

Table 1　Results of DP-LS in comparison with the reference algorithms (i.e., GRASP, TS+PR, and Gurobi) for the 22 typical instances

n_1	n_2	Density	γ	Best known	GRASP			TS+PR			Gurobi			DP-LS		
					Cross	Time (s)	GAP (%)	Cross	Time (s)	GAP (%)	Cross	Time (s)	GAP (%)	Cross	Time (s)	GAP (%)
25	25	0.065	1.2	305	316	6.1	3.6	**305**	0.4	0	**305**	0.4	0	**305**	0.01	0
25	25	0.065	1.6	979	1131	34.5	15.5	1106	1.8	13	**979**	237.7	0	980	1.17	0.1
25	25	0.175	1.2	3922	4037	36.9	2.9	3927	0.4	0.1	**3922**	1.4	0	**3922**	0.01	0
25	25	0.175	1.6	11982	12374	120	3.3	11982	3.8	0	12444	1800.6	3.9	**11794**	0.18	−1.57
25	25	0.3	1.2	15036	15185	60.9	1	15067	0.6	0.2	**15036**	9.6	0	**15036**	0	0
25	25	0.3	1.6	39657	40538	252.2	2.2	39657	7.7	0	41705	1801.6	5.2	**39465**	0.11	−0.48
25	50	0.065	1.2	2173	2218	27.8	2.1	2185	0.7	0.6	**2173**	1.5	0	**2173**	0.04	0
25	50	0.175	1.2	19794	20112	201.7	1.6	19861	2.1	0.3	**19794**	193.3	0	**19794**	0.01	0
25	50	0.3	1.2	62986	64113	495.3	1.8	63312	4.8	0.5	62986	1802.3	0	**61813**	0.02	−1.86
25	50	0.3	1.6	175764	175764	519	0	176909	52.8	0	184788	1831.6	4.5	**172837**	0.04	−1.67
50	25	0.065	1.2	2169	2230	42.2	2.8	2191	0.8	1	**2169**	1.6	0	**2169**	0.03	0
50	25	0.065	1.6	5828	6459	267.3	10.8	5828	12.4	0	6155	1801	5.6	**5552**	0.25	−4.74
50	25	0.175	1.2	19831	20265	228.9	2.2	19890	2	0.3	**19831**	336.2	0	**19831**	0.03	0
50	25	0.175	1.6	54004	57004	517.7	5.6	54004	31.3	0	63287	1802.4	17.2	**53396**	0.26	−1.13
50	25	0.3	1.2	65593	66253	502.8	1	65593	4.4	0	66319	1802	1.1	**65557**	0.04	−0.05
50	25	0.3	1.6	179282	186429	538.6	4	179282	58.6	0	189119	1805.3	5.5	**176557**	0.23	−1.52
50	50	0.065	1.2	7637	7859	297.3	2.9	7664	3.9	0.4	**7637**	3.5	0	**7637**	0.01	0
50	50	0.065	1.6	24933	25545	506.5	2.5	24933	67	0	27110	1802.8	8.7	**24103**	6.66	−3.33
50	50	0.175	1.2	77253	78717	505.5	1.9	77253	14.3	0	77480	1802.7	0.3	**77205**	0.02	−0.06
50	50	0.175	1.6	233326	238979	504.1	2.4	233326	209.3	0	256917	1808.6	10.1	**230789**	2.27	−1.09
50	50	0.3	1.2	248454	251277	536.8	1.1	248454	26.1	0	258330	1806.3	4	**248172**	0.06	−0.11
50	50	0.3	1.6	712459	728794	575.7	2.3	712459	416.7	0	757750	1800.3	6.4	**709286**	1.44	−0.45
#Avg				89243.95	91163.59	308.08	3.34	89326.72	41.90	0.75	94374.36	1102.40	3.30	**88562.41**	**0.59**	**−0.82**
#Best					0			1			9			**21**		
p-value					2.73e−06			4.59e−06			1.35e−3					

表 2　DP-LS 算法和算法 TS+PR 以及通用求解器 Gurobi 在 9 个大规模算例 (包含 471 个节点) 上的实验结果

Table 2　Results of DP-LS in comparison with algorithm TS+PR and the general solver Gurobi on 9 large-scale instances with 471 vertices

Instance name	Best known	TS+PR			Gurobi			DP-LS		
		Cross	Time (s)	GAP (%)	Cross	Time (s)	GAP (%)	Cross	Time (s)	GAP (%)
G_00_05_scr_0014_10	47050	47587	524.49	1.14	47050	1829.01	0	**46796**	1.34	−0.54
G_00_05_scr_0014_20	39523	41740	578.29	5.61	39523	1827.62	0	**38235**	7.06	−3.26
G_00_05_scr_0014_30	36446	36446	710.91	0	48446	1830.41	32.93	**32201**	15.89	−11.65
G_00_05_scr_0015_10	45343	45803	534.58	1.01	45343	1864.35	0	**45016**	0.21	−0.72
G_00_05_scr_0015_20	40142	40142	582.47	0	47580	1836.25	18.53	**37629**	6.65	−6.26
G_00_05_scr_0015_30	36948	36948	679.92	0	46733	1840.15	26.48	**32256**	22.29	−12.7
G_00_05_scr_0016_10	44822	44822	512.09	0	53518	1832.23	19.4	**43228**	0.20	−3.56
G_00_05_scr_0016_20	42348	42348	575.57	0	47913	1836.74	13.14	**39222**	16.87	−7.38
G_00_05_scr_0016_30	37349	37349	685.11	0	47236	1835.91	26.47	**31925**	16.09	−14.52
#Avg	41107.89	41465	598.16	0.86	47038	1836.96	15.22	**38500.89**	**9.63**	**−6.73**
#Best		0			0			**9**		
p-value		2.7e−3			2.7e−3					

表 3　与 4 个参考算法 (GRASP, TS(500)+PR, TS+PR 和 Gurobi) 基于第 1 组算例集的实验结果对比

Table 3　Results of DP-LS in comparison with the reference algorithms (i.e., GRASP, TS(500)+PR, TS+PR, and Gurobi) on the first instance set with $\gamma = 1.2$ and 1.6

Size $(n_1 + n_2)$	Algorithm	$\gamma = 1.2$			$\gamma = 1.6$			#Best
		ACross	ATime (s)	AGP (%)	ACross	ATime (s)	AGP (%)	
50	Gurobi	6230.00	27.18	0.00	18553.53	1224.33	4.24	–
	GRASP	6317.47	33.23	2.17	18056.27	141.83	7.44	
	TS(500)+PR	6232.20	48.76	0.10	17459.13	1039.53	1.01	
	TS+PR	6234.33	8.15	0.16	17489.07	90.12	1.71	
	DP-LS	**6230.00**	**0.03**	**0.00**	**17376.8**	**0.74**	**0.54**	
75	Gurobi	28522.20	781.71	0.24	84088.57	1808.00	7.50	–
	GRASP	28776.57	228.98	2.52	80917.97	409.31	5.15	
	TS(500)+PR	28379.10	621.53	0.16	78959.23	1718.27	0.27	
	TS+PR	28390.17	61.11	0.21	79161.07	386.90	0.61	
	DP-LS	**28359.93**	**0.02**	**0.09**	**77791.53**	**1.24**	**−1.21**	
100	Gurobi	112572.27	1207.01	1.10	343478.67	1891.89	8.70	–
	GRASP	111381.07	423.60	1.94	328878.53	532.43	3.86	
	TS(500)+PR	110214.33	1516.66	0.08	322528.33	1687.68	0.00	
	TS+PR	110233.07	292.30	0.12	322831.60	514.51	0.23	
	DP-LS	**110172.60**	**0.41**	**−0.04**	**319628.67**	**3.97**	**−0.89**	
#Total	TS(500)+PR	83888.83	1121.52					7 (120)
	DP-LS	**83213.88**	**0.96**					**119 (120)**

表 4 DP-LS 算法与两个参考算法 (GRASP 和 TS+PR) 基于第 2 组算例集的比较结果

Table 4 Performance results of DP-LS in comparison with the reference algorithms (i.e., GRASP and TS+PR) on the second set of instances

Size	Algorithm	ACross	ATime (s)	AGP (%)	#Best
$21 \leqslant n_1 + n_2 \leqslant 57$	GRASP	510.82	0.40	4.09	–
	TS+PR	497.77	0.09	2.03	
	DP-LS	**487.79**	**0.02**	**−2.01**	
$111 \leqslant n_1 + n_2 \leqslant 122$	GRASP	22742.99	18.83	4.69	–
	TS+PR	22394.12	7.80	0.06	
	DP-LS	**22327.91**	**0.11**	**−0.30**	
$231 \leqslant n_1 + n_2 \leqslant 471$	GRASP	87844.49	319.53	4.97	–
	TS+PR	85816.74	232.27	0.01	
	DP-LS	**84850.02**	**2.48**	**−1.13**	
#Total	TS+PR	36928.31	83.67		241 (1000)
	DP-LS	**36566.61**	**0.91**		**990 (1000)**

间的平均 GAP 值 (AGP(%)) 3 个指标上全面超越了其他 4 个算法. 对于总节点数为 50 的小规模算例, DP-LS 算法和 Gurobi 求解器均能找到该组所有算例的最优解, 但在计算时间上, DP-LS 算法要明显快于 Gurobi (0.03 s vs. 27.18 s). 随着算例规模的增大 (即总节点数的增加), DP-LS 算法的优势愈加明显. 在中等规模 (总结点数分别为 75 和 100) 算例上, DP-LS 算法均能以更快的速度比其他算法找到更好的解. 综合第 1 组算例集的实验结果, DP-LS 算法比对比算法中最好的 TS(500)+PR 算法具有更好的解的优度 (83213.88 vs. 83888.83). 同时, DP-LS 算法能够找到 120 个算例中 119 个算例的最好解, 而算法 TS(500)+PR 只能找到 7 个算例的最好解. 在平均计算时间方面, DP-LS 较 TS(500)+PR 快了接近 1000 倍 (0.96 s vs. 1121.52 s). 因此, 本节基于第 1 组算例集的实验结果表明了 DP-LS 算法在解的优度和计算效率两方面的有效性和高效性.

4.4 第 2 组算例集的计算结果

本小节针对第 2 组算例集, 选取文献中算法性能表现最好的两种 (GRASP 和 TS+PR 算法) 作为比较算法进行本节的性能对比实验. TS(500)+PR 和 Gurobi 算法在之前文献中没有报道第 2 组算例集的实验结果. 表 4 报道了 3 种算法在 3 种不同规模的算例上的计算结果对比. 其中, 小规模算例的总节点数介于 [27, 57] 之间, 中等规模算例的总节点数介于 [111, 122] 之间, 大规模算例的总节点数介于 [231, 471] 之间. 从表 4 可以看出, DP-LS 算法在这 3 个不同规模的算例子集上都能得到比另外两个对比算法更好的解. 同时, 在计算时间方面, 随着问题规模的增大, GRASP 和 TS+PR 算法的计算时间呈现出明显增加的趋势. 而 DP-LS 算法即使在最大规模的一组算例上依然可以以平均 2.48 s 的速度找到问题的最好解. 对于第 2 组算例集共 1000 个算例, DP-LS 算法能够找到其中 990 个算例的最好解, 而 TS+PR 算法只能找到其中 241 个算例的最好解. 基于本组共 1000 个算例的实验结果, DP-LS 算法较 TS+PR 算法在计算时间上快了数十倍 (0.91 s vs. 83.67 s). 综上所述, 基于两组算例集共 1120 个算例的实验结果, DP-LS 算法不仅能够在解的优度上优于其他启发式算法, 而且在计算时间上也具有明显的优势, 表明了 DP-LS 算法在求解 DBDP 问题上的高效性.

5 策略有效性和参数敏感性分析

5.1 动态规划机制的有效性分析

文献 [9∼13] 中几乎所有求解 DBDP 问题的邻域搜索算法在每次邻域评估时, 只挑选一个最好的邻域动作并执行. 本文提出了一个动态规划驱动的局部搜索算法, 能够同时从邻域中挑选出多个独立的邻域动作. 为了检验该策略的有效性, 本文选取了一个未采用动态规划机制的迭代局部搜索算法 ILS 作为 DP-LS 的参考算法进行对比实验. 未采用动态规划机制的 ILS 算法采用与之前文献中相同的邻域搜索技术, 即在每次邻域评估中选取邻域内最好的一个邻域动作进行执行. 如果存在多个最好的邻域动作, 则进行随机选择. ILS 算法的初始解生成算法、扰动策略, 以及算法终止条件均与 DP-LS 算法一致.

表 5 报道了基于 22 个测试算例集的测试结果, 其中, DP-LS 算法在找到解的交叉边数 Cross 和计算时间 Time (s) 方面较 ILS 算法有明显的优势 (88562.41 vs. 89253.64 以及 0.59 s vs. 20.23 s). 在找到的最好解个数 (#Best) 指标上, DP-LS 能找到 22 个算例中 21 个算例的最好结果, 而 ILS 算法仅能找到 10 个算例的最好解. 最后一行给出了 DP-LS 算法和 ILS 算法基于非参数 Friedman 测试的计算结果. 小于 0.05 的 p-value 值 (2.73e−06) 表明了 DP-LS 算法和 ILS 算法在本组算例集上具有显著性差异.

表 6 统计了所有 1120 个算例的实验结果. 其中, 第 1 和 2 列分别表示算例所属算例集以及总的节点个数. DP-LS 算法找到的平均值为 41564.52, 要明显地优于 ILS 的 41669.01, 同时 DP-LS 只需要 0.91 s 的计算时间, 明显地快于 ILS 算法的 8.32 s. 因此, 本实验表明了 DP-LS 算法中动态规划机制的有效性和高效性.

5.2 邻域动作分析

为了增强算法效率, 本文提出了基于两种邻域动作 (插入和交换动作) 的局部搜索方法. 为了验证这两种邻域动作的有效性, 本小节选取了 DP-LS 算法的两个退化版本 (只包含插入动作的版本 DP-LS$_{\text{insert}}$ 和只包含交换动作的版本 DP-LS$_{\text{swap}}$) 来进行对比实验.

表 7 的实验结果表明, 基于两种邻域动作的标准 DP-LS 算法在解的平均优度和找到最好解 (3 个对比算法中的最好解) 的个数上都要优于仅使用插入动作的 DP-LS$_{\text{insert}}$ (41564.52 vs. 41566.01 以及 1098 vs. 1041). 而在计算时间方面, DP-LS$_{\text{insert}}$ 要略差于 DP-LS (0.91 s vs. 0.74 s). 值得注意的是, 仅使用交换动作的 DP-LS$_{\text{swap}}$ 表现最差, 其性能表现全面落后于另外两种算法. 因此, 该实验结果表明对于 DBDP 问题和 DP-LS 算法, 插入动作要比交换动作更加有效, 而将两种邻域动作进行组合同时使用则能进一步提升算法效率.

5.3 参数敏感性分析

本小节通过计算实验来分析 DP-LS 算法中两个参数的不同取值对算法性能的影响.

• 算法终止参数 Θ. 表示历史最优解在多少迭代步数内无法改进则算法终止. 在本实验中, 对比的 3 个 Θ 值为: 5000, 10000, 20000.

• 扰动参数 α. 表示扰动中删除增量节点的比率 (删除节点个数为 $\alpha \times |\text{AV}|$). 在本实验中, 对比的 3 个 α 值为: 0.2, 0.5, 0.8.

参数性能对比的结果如表 8 所示. 其中, ACross 和 ATime (s) 列分别表示基于第 1 组算例的平均交叉数和平均计算时间, #Best 和 AGP (%) 分别表示找到最好解的数量以及和最好解之间的平

表 5 DP-LS 算法同未采用动态化规划驱动的局部搜索 ILS 基于 22 个测试算例集的实验结果

Table 5 Results of DP-LS in comparison with the reference algorithm ILS without dynamic programming mechanism on the 22 typical instances

n_1	n_2	Density	γ	Best known	DP-LS			ILS		
					Cross	Time (s)	GAP (%)	Cross	Time (s)	GAP (%)
25	25	0.065	1.2	305	305	0.01	0	305	0.05	0
25	25	0.065	1.6	979	980	1.17	0.1	1006	1.54	2.76
25	25	0.175	1.2	3922	3922	0.01	0	3922	0.54	0
25	25	0.175	1.6	11982	11794	0.18	−1.57	12417	6.95	3.63
25	25	0.3	1.2	15036	15036	0.01	0	15036	0.86	0
25	25	0.3	1.6	39657	39465	0.11	−0.48	39597	18.96	−0.15
25	50	0.065	1.2	2173	2173	0.04	0	2173	0.47	0
25	50	0.175	1.2	19794	19794	0.01	0	19794	2.20	0
25	50	0.3	1.2	62986	61813	0.02	−1.86	62794	7.65	−0.3
25	50	0.3	1.6	175764	172837	0.04	−1.67	172841	56.93	−1.66
50	25	0.065	1.2	2169	2169	0.03	0	2169	0.48	0
50	25	0.065	1.6	5828	5552	0.25	−4.74	5699	7.39	−2.21
50	25	0.175	1.2	19831	19831	0.03	0	19831	4.62	0
50	25	0.175	1.6	54004	53396	0.26	−1.13	53387	48.09	−1.14
50	25	0.3	1.2	65593	65557	0.04	−0.05	65557	8.40	−0.05
50	25	0.3	1.6	179282	176557	0.23	−1.52	184608	33.48	2.97
50	50	0.065	1.2	7637	7637	0.01	0	7637	1.523	0
50	50	0.065	1.6	24933	24103	6.66	−3.33	24808	19.33	−0.5
50	50	0.175	1.2	77253	77205	0.02	−0.06	77205	12.59	−0.06
50	50	0.175	1.6	233326	230789	2.27	−1.09	232465	102.41	−0.37
50	50	0.3	1.2	248454	248172	0.06	−0.11	248172	49.09	−0.11
50	50	0.3	1.6	712459	709286	1.44	−0.45	712157	61.58	−0.04
#Avg				**89243.95**	**88562.41**	**0.59**	**−0.82**	89253.64	20.23	0.13
#Best					**21**			10		
p-value					2.73e−06					

均百分比差. 通过观察发现, 平均目标函数值 ACross 表现最好的 3 个参数组合分别为: (0.8, 20000), (0.5, 20000), 以及 (0.5, 10000). 由于前两个参数组合 (0.8, 20000) 和 (0.5, 20000) 其对应的平均计算时间 ATime (s) 表现最差. 因此, 本实验的标准算法选取了能在平均交叉数 ACross 以及平均计算时间 ATime(s) 上取得更好平衡的参数组合 (0.5, 10000). 同时, 该参数组合在 #Best 和 AGP (%) 指标上也表现较好. 此外, 比较接近的实验结果也说明了参数的不同取值对算法性能的影响不大.

6 结论

增量二分绘图问题 (DBDP) 是一个具有 NP 难度的组合优化问题, 该问题在现实生活中有着广泛的应用. 本文提出了一种新的动态规划驱动的局部搜索 (DP-LS) 算法来求解该问题. 该方法能够能从邻域结构中挑选出多个独立的动作并同时执行, 大大提升了算法的搜索效率. 通过与文献中最好的多

表 6 DP-LS 算法同未采用动态规划驱动的局部搜索 ILS 基于所有 1120 个算例的实验结果

Table 6 Performance results of DP-LS in comparison with the reference algorithm ILS without dynamic programming mechanism on both two instance sets

Instance set	n_1+n_2	DP-LS			ILS		
		ACross	ATime(s)	#Best	ACross	ATime(s)	#Best
1	50	11803.4	0.38	30(30)	11963.3	5.74	15(30)
1	75	53075.73	0.63	60(60)	53531.88	16.32	34(60)
1	100	214900.63	2.19	30(30)	215631.1	52.53	13(30)
2	21	49.28	0.01	150(150)	51.01	0.02	104(150)
2	51	372.36	0.04	119(150)	380.25	0.02	31(150)
2	57	2149.6	0.02	50(50)	2155.52	0.09	41(50)
2	111	2002.88	0.09	150(150)	2004.96	0.28	56(150)
2	122	42652.94	0.12	119(150)	42709.43	4.76	110(150)
2	231	9254.73	2.07	145(150)	9269.93	6.89	102(150)
2	242	175692.80	0.43	115(150)	175982.24	8.03	90(150)
2	471	39107.56	9.82	43(50)	39241.92	19.47	14(50)
#Total		**41564.52**	**0.91**	**1011(1120)**	41669.01	8.32	610(1120)

表 7 DP-LS 算法同只包含插入或交换动作的弱化版本 (DP-LS$_{\text{insert}}$ 和 DP-LS$_{\text{swap}}$) 基于所有 1120 个算例的实验结果

Table 7 Performance results of DP-LS in comparison with two variations (i.e., DP-LS$_{\text{insert}}$ and DP-LS$_{\text{swap}}$) only including insert moves or swap moves on all the 1120 instances

Instance set	n_1+n_2	DP-LS			DP-LS$_{\text{insert}}$			DP-LS$_{\text{swap}}$		
		ACross	ATime (s)	#Best	ACross	ATime (s)	#Best	ACross	ATime (s)	#Best
1	50	11803.4	0.38	30(30)	11803.5	0.42	28(30)	12092.17	6.91	0(30)
1	75	53087.95	0.63	58(60)	53081.35	0.57	52(60)	53778.35	11.64	0(30)
1	100	214900.63	2.19	29(30)	214902.17	1.15	20(30)	216644.9	16.34	1(30)
2	21	49.28	0.01	150(150)	49.3	0.01	148(150)	55.62	0.28	32(150)
2	51	372.36	0.04	148(150)	373.27	0.01	150(150)	411.13	4.43	1(150)
2	57	2149.6	0.02	49(50)	2149.5	0.01	50(50)	2287.42	5.99	0(50)
2	111	2002.88	0.09	150(150)	2002.88	0.15	150(150)	2001.86	21.94	0(150)
2	122	42652.94	0.12	145(150)	42654.05	0.09	143(150)	43852.7	0.92	4(150)
2	231	9254.73	2.07	149(150)	9255.1	2.49	126(150)	10148.98	30.54	0(150)
2	242	175692.8	0.43	147(150)	175696.98	0.40	144(150)	179762.27	10.67	1(150)
2	471	39107.56	9.82	43(50)	39113.3	5.55	30(50)	43087	31.54	0(50)
#Total		**41564.52**	0.91	**1098(1120)**	41566.01	**0.74**	1041(1120)	42698.59	13.47	39(1120)

种算法以及通用求解器 Gurobi 的对比实验, 表明了提出的 DP-LS 算法在求解 DBDP 问题上的高效性. 在两组共 1120 个算例上较之前文献中最好的算法 (禁忌搜索和路径重连算法) 能以更快的速度找

表 8 参数性能对比

Table 8 Performance comparison of different parameter settings

Parameters		Average performance			
α	Θ	ACross	ATime (s)	#Best	AGP (%)
0.2	5000	83227.74	0.21	108	4.64
0.2	10000	83221.72	0.29	111	3.59
0.2	20000	83221.00	0.50	114	3.25
0.5	5000	83210.69	0.43	112	0.34
0.5	10000	83210.61	0.61	113	0.30
0.5	20000	83210.52	1.08	115	0.19
0.8	5000	83211.19	0.60	103	0.66
0.8	10000	83212.23	0.89	108	0.51
0.8	20000	83210.59	1.65	114	0.10

到更高质量的解. 对于未来的研究展望, 本文希望将提出的 DP-LS 算法拓展到增量多层图问题以及其他排序类优化问题上去.

致谢 感谢 Rafael Martí 教授为本文提供测试算例和对比算法的实验数据.

参考文献

1 Antonellis I, Molina H G, Chang C C. Simrank++: query rewriting through link analysis of the click graph. Proc VLDB Endow, 2008, 1: 408–421

2 Even G, Guha S, Schieber B. Improved approximations of crossings in graph drawings and VLSI layout areas. SIAM J Comput, 2003, 32: 231–252

3 Oselio B, Kulesza A, Hero A O. Multi-layer graph analytics for social networks. In: Proceedings of the 5th IEEE International Workshop on Computational Advances in Multi-Sensor Adaptive Processing (CAMSAP), 2013. 284–287

4 Burch M, Müller C, Reina G, et al. Visualizing dynamic call graphs. 2012. http://www.profitippliga.de/papers/31.pdf

5 Erten C, Harding P J, Kobourov S G, et al. Graphael: graph animations with evolving layouts. In: Proceedings of International Symposium in Graph Drawing, 2003. 98–110

6 Garey M R, Johnson D S. Crossing Number is NP-Complete. SIAM J Algebr Discrete Meth, 1983, 4: 312–316

7 Eades P, Kelly D. Heuristics for drawing 2-layered networks. Ars Combin, 1986, 21: 89–98

8 Jünger M, Mutzel P. 2-layer straightline crossing minimization: performance of exact and heuristic algorithms. J Graph Alg Appl, 1997, 1: 1–25

9 Martí R, Estruch V. Incremental bipartite drawing problem. Comput Oper Res, 2001, 28: 1287–1298

10 Martí R, Martínez-Gavara A, Sánchez-Oro J, et al. Tabu search for the dynamic bipartite drawing problem. Comput Oper Res, 2018, 91: 1–12

11 Martí R, Campos V, Hoff A, et al. Heuristics for the min-max arc crossing problem in graphs. Expert Syst Appl, 2018, 109: 100–113

12 Napoletano A, Martínez-Gavara A, Festa P, et al. Heuristics for the constrained incremental graph drawing problem. Eur J Oper Res, 2019, 274: 710–729

13 Sánchez-Oro J, Martínez-Gavara A, Laguna M, et al. Variable neighborhood scatter search for the incremental graph drawing problem. Comput Optim Appl, 2017, 68: 775–797

14 Wang Z, Lü Z P, Ye T. Local search algorithms for large-scale load balancing problems in cloud computing. Sci Sin Inform, 2015, 45: 587–604 [王卓, 吕志鹏, 叶涛. 求解大规模云计算负载均衡问题的局部搜索算法. 中国科学: 信息科学, 2015, 45: 587–604]

15 Cai S W, Luo C, Lin J K, et al. New local search methods for partial MaxSAT. Artif Intell, 2016, 240: 1–18
16 Cai S W, Hou W Y, Lin J K, et al. Improving local search for minimum weight vertex cover by dynamic strategies. In: Proceedings of International Joint Conference on Artificial Intelligence, 2018. 1412–1418
17 Lü Z, Hao J K. A memetic algorithm for graph coloring. Eur J Oper Res, 2010, 203: 241–250
18 Stallmann M, Brglez F, Ghosh D. Heuristics, experimental subjects, and treatment evaluation in bigraph crossing minimization. J Exp Algorithm, 2001, 6: 8–42

A dynamic programming-based local search algorithm for solving the dynamic bipartite drawing problem

Bo PENG[1], Chenbei LU[2], Yuehu ZHAO[2], Zhouxing SU[2], Yi LIAO[1] & Zhipeng LÜ[2*]

1. *School of Business Administration, Southwestern University of Finance and Economics, Chengdu* 610074*, China*;
2. *Laboratory of Smart Computing and Optimization, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan* 430074*, China*
* Corresponding author. E-mail: zhipeng.lv@hust.edu.cn

Abstract The dynamic bipartite drawing problem is a challenging NP-hard combinatorial optimization problem with numerous applications. In this study, we propose a dynamic programming-based local search (DP-LS) algorithm for solving it. Unlike previous metaheuristics reported in the literature, which move just one vertex (or two vertices) at each neighborhood iteration, the proposed DP-LS algorithm selects and performs multiple independent neighborhood moves at the same time to enhance the search efficiency. Generally, starting from a random initial solution, DP-LS iteratively explores the search space by integrating the DP-LS to locate local optima, and then uses a perturbation procedure to escape from the local optima. In addition, the proposed incremental evaluation techniques of the insert and swap moves enhance the efficiency of the neighborhood evaluation. Extensive computational experiments on two sets of 1120 problem instances indicate that the proposed DP-LS is highly competitive with the best-performing algorithms (including the general purpose solver Gurobi) in terms of both solution quality and computational efficiency. We analyzed the dynamic programming mechanism in the proposed DP-LS algorithm to determine its effectiveness (increasing the search efficiency tens of times). Not only can the proposed DP-LS algorithm be used to solve the DBDP, it can also be used as a general methodology for solving other combinatorial optimization problems, especially permutation optimization problems.

Keywords dynamic bipartite drawing problem, dynamic programming, local search, incremental evaluation technique

Bo PENG was born in 1988. He received a Ph.D. degree in computer science from Huazhong University of Science and Technology, China, in 2017. Currently, He is a lecturer at the School of Business and Administration at Southwestern University of Finance and Economics. His research interests include artificial intelligence, computational intelligence, and adaptive metaheuristics for solving hard combinatorial optimization problems.

Yuehu ZHAO was born in 1998. He is an undergraduate student in industrial engineering at Huazhong University of Science and Technology, China. His research interests include flow shop scheduling and graph drawing problems.

Chenbei LU was born in 1998. He is an undergraduate student at Huazhong University of Science and Technology, China. His research interests include artificial intelligence, meta-heuristic algorithms for large-scale theoretical combinatorial optimization problems, and object detection based on deep neural networks.

Zhipeng LÜ was born in 1979. He received a B.S. degree in applied mathematics from Jilin University, China, in 2001, and a Ph.D. degree in computer software and theory from Huazhong University of Science and Technology, China, in 2007. He was a postdoctoral research fellow at LERIA, Department of Computer Science, University of Angers, France, from 2007 to 2011. He is a professor at the School of Computer Science and Technology of Huazhong University of Science and Technology, and is the director of the Laboratory of Smart Computing and Optimization (SMART). His research is in the area of artificial intelligence, computational intelligence, green computing, and adaptive metaheuristics for solving large-scale real-world and theoretical combinatorial optimization and constrained satisfaction problems.