中国科学: 技术科学
SCIENTIA SINICA Technologica
techcn.scichina.com
《中国科学》杂志社
SCIENCE CHINA PRESS
论 文 群体智能激发汇聚及应用专辑

# 近似最优并行算法组智能汇聚构造

刘晟材[1,2], 杨鹏[1,2], 唐珂[1,2*]

1. 南方科技大学计算机科学与工程系, 深圳 518055;
2. 广东省类脑智能计算重点实验室, 深圳 518055
*E-mail: tangk3@sustech.edu.cn

收稿日期: 2021-08-14; 接受日期: 2021-10-14; 网络版发表日期: 2022-08-03
广东省类脑智能计算重点实验室(编号: 2020B121201001)和国家自然科学基金青年科学基金(批准号: 61806090)资助项目

**摘要** 作为一种高性能通用并行求解器, 并行算法组(parallel algorithm portfolios, PAPs)近年来在判定、计数以及连续、离散优化等问题上取得了突出的求解效果. 传统人工构造PAP的方式依赖于大量领域知识, 门槛极高. 为了解决这一问题, 本文提出了一种基于演化优化的PAP智能汇聚自动构造方法AutoPAP. 整体上, AutoPAP遵循($n$+1)演化优化框架, 即在每一代生成$n$个候选算法, 并保留最优算法加入到PAP中. 考虑到算法配置空间往往非常巨大且涉及混合变量, 本文设计了专用变异算子以提升AutoPAP的实际性能, 并证明了AutoPAP在理论上可以达到$(1-1/e)$近似最优构造效果. 最后, 本文以旅行商问题(traveling salesman problems, TSP)为例, 使用AutoPAP构造得到TSP_PAP. 实验结果表明, 在主流TSP测试集上, TSP_PAP的求解效率和效果均显著好于当前TSP上公认性能最佳的求解器EAX和LKH. 在128个规模1000~30000的TSP测试样例上, 相比于EAX和LKH, TSP_PAP可以将平均求解时间缩短至少45.71%, 并将平均误差率降低至少87.50%, 这体现出AutoPAP在算法自动设计与演化方面的巨大潜力.

**关键词** 求解器, 并行算法组, 演化计算, 自动算法设计, 旅行商问题, 组合优化

## 1 引言

过去20年间, 并行计算架构得到了巨大发展[1]. 现如今多核中央处理器(central processing unit, CPU)已然成为个人计算机的标准配置, 而在那些专为科学计算而搭建的计算平台如工作站和服务器上, CPU核数动辄几十上百. 如此丰富的计算资源也给算法/求解器设计者们提出了新的挑战: 如何有效利用并行计算平台以更好地解决实际应用中的复杂问题? 事实上, 无论是学术界还是工业界, 对于并行求解器的研究已持续多年. 例如, 在一些重要的基本计算问题如布尔可满足性问题(boolean satisfiability problem, SAT)、约束满足问题(constraint satisfaction problem, CSP)、回答集编程问题(answer set programming, ASP)、混合线性整数规划问题(mixed integer linear programming, MILP)和黑箱连续优化问题(black-box continuous optimization, BO)上, 并行求解器[2~6]已经成为主流. 此外, 在很多基础工业软件如数学规划求解器CPLEX

引用格式: 刘晟材, 杨鹏, 唐珂. 近似最优并行算法组智能汇聚构造. 中国科学: 技术科学, 2022, 52
Liu S C, Yang P, Tang K. Approximately optimal construction of parallel algorithm portfolios by evolutionary intelligence (in Chinese). Sci Sin Tech, 2022, 52, doi: 10.1360/SST-2021-0372

(https://www.ibm.com/analytics/cplex-optimizer)和EDA软件Synopsys(https://www.synopsys.com)中, 并行求解器已经成为其核心模块.

虽然并行求解器已取得一定程度的发展, 其仍面临着研发难度高、周期长的困境. 一般而言, 并行求解器的研发始于改造已有串行求解器, 并需进一步集成信息交互机制以实现问题分解和不同求解线程之间的协作[7]. 这一流程要求研发人员具备大量相关领域知识, 并需要其耗费大量时间对求解器进行迭代改进, 耗时耗力.

近年来, 一种新的并行求解器形式——并行算法组(parallel algorithm portfolios, PAP), 在判定[8]、计数以及连续[6]、离散优化[8~10]等问题上取得了突出的求解效果, 逐渐成为了研究热点. 本质上PAP是一个包含若干算法的集合, 其中每个算法都被称作该PAP的成员算法. 当PAP被用于求解某个问题样例时, 其所有成员算法都将在该问题样例上相互独立地并行运行, 且在达到停止条件时同时终止. 可以看到, 在PAP运行过程中, 各个成员算法之间不涉及任何信息交互, 因而PAP的结构复杂度相比于传统并行求解器要低得多.

另一方面, 构造高性能PAP并不容易. 假设一个PAP的某个成员算法$\theta$在任何问题样例上的性能都超过了其他成员算法, 那么该PAP的性能实际上仅等价于算法$\theta$的性能. 换言之, 除$\theta$之外的成员算法虽然使用了与$\theta$相同的计算资源, 却并未给PAP带来任何性能增益. 因此, 直观上而言, 一个高性能PAP所包含的成员算法在性能上应具备差异性, 即各个成员算法都应该有自身最为擅长解决的问题样例类型. 可以说, 认识到这种差异性是构造高性能PAP的前提条件. 然而, 在实际应用中找到符合以上条件的算法并不容易, 这要求研发人员对各种算法的优势以及短板有充分认识, 且需耗费大量时间进行后续调整. 这不仅带来了高昂的人力成本, 也在无形中提高了PAP的研发门槛, 阻碍其进一步发展.

为了解决以上问题, 本文提出了一种基于$(n+1)$演化优化框架的PAP智能汇聚构造方法AutoPAP, 其接受一个算法配置空间和一个问题样例集合作为输入, 从前者中自动为PAP选择成员算法, 以使得PAP在后者上的性能达到最优. 为了提升AutoPAP的实际应用效果, 本文结合自动算法配置技术和分而治之策略设计了可以高效搜索算法配置空间的变异算子. 进一步地, 本文从理论上证明了AutoPAP可以达到$(1-1/e)$的近似最优比.

最后, 本文以旅行商问题(traveling salesman problems, TSP)为例, 验证AutoPAP的有效性. 具体而言, 本文使用AutoPAP构造得到TSP_PAP, 并将其和当前TSP上性能最佳的求解器LKH[11]以及EAX[12]作对比. 实验结果表明, 在主流TSP测试集上, TSP_PAP无论是在求解效率还是在求解效果上均显著好于LKH和EAX. 在128个规模1000~30000的TSP测试样例上, 相比于LKH和EAX, TSP_PAP可以将平均求解时间缩短至少45.71%, 并将平均误差率降低至少87.50%. 可以说, TSP_PAP是当前综合性能最强的TSP求解器, 这体现出AutoPAP在算法自动设计与演化方面的巨大潜力.

本文的后续章节安排如下. 第二节分别给出PAP以及PAP自动构建问题的形式化定义. 第三节首先描述AutoPAP框架以及变异算子, 然后证明AutoPAP的近似最优性. 第四节以TSP问题为例, 验证AutoPAP的有效性. 第五节对全文进行总结.

## 2 问题定义

### 2.1 PAP形式化定义

令$P$表示PAP, 令$k$表示$P$的规模, 即成员算法的数量, $P$的形式化定义如下:

$$P=\{\theta_1,\theta_2,...,\theta_k\}, \tag{1}$$

式中, $\theta_i$表示$P$中第$i$个成员算法.

为了避免引入过多的数学符号, 本文以m(solver, instances)表示在性能指标m下, 求解器“solver”在问题样例集合“instances”上的性能. 注意“solver”可以是单个算法$\theta$, 也可以是一个PAP, 而“instances”既可以是单个问题样例(此时可以看作只含有一个样例的集合), 也可以是包含多个问题样例的集合.

当使用$P$来求解给定问题样例$z$时, $P$中所有成员算法, 即$\theta_1,\theta_2,...,\theta_k$, 都将在$z$上相互独立地并行运行, 直到达到终止条件. 这里的终止条件依赖于待求解的问题和感兴趣的性能指标m. 具体而言, 假设待求解的问题为判定类(decision)问题(如SAT问题), 那么在PAP

运行过程中只要任何一个成员算法率先输出了对$z$的判定结果, 即“是”或“否”, 所有成员算法都会立即被终止. 因此求解$z$所需的运行时间就是其最佳成员算法求解$z$所需的运行时间. 此外, 在这种情况下, 通常会引入一个最长运行时间(又称截止时间, $T_{\max}$)以防止PAP运行时间过长. 如果在$T_{\max}$时间内, 没有任何成员算法输出判定答案, 那么所有成员算法也会被立即终止, 且此次求解被判定为超时(Timeout)或失败(Failure).

另一方面, 假设待求解的问题为优化问题(如TSP问题), 终止条件则依赖于感兴趣的性能指标m. 如果m是“找到一个近似最优解(如与最优解的差距不超过预先给定的阈值)所需的运行时间”, 那么只要任何一个成员算法率先找到了这类解, 所有成员算法都会立即被终止. 与考虑决策问题时一样, 在这种情况下可以引入最长运行时间$T_{\max}$以防止PAP运行时间过长. 如果m是“在给定时间$T_{\max}$内找到的解的质量”, 那么每个成员算法都将在运行$T_{\max}$时间后终止, 最后$P$将成员算法找到的所有解中最好的那个作为输出.

可以看到, 无论考虑何种问题类型和何种性能指标, $P$在问题样例$z$上的性能$\mathrm{m}(P,z)$总是其成员算法$\theta_1,\theta_2,...,\theta_k$在$z$上取得的最好性能, 即

$$\mathrm{m}(P,z)=\max_{\theta\in P}\mathrm{m}(\theta,z). \tag{2}$$

不失一般性, 假设对性能指标m而言, 值越大越好. 值得注意的是, 实际中考虑优化问题且m是“找到一个近似最优解所需的运行时间”时, 我们往往并不知道待求解问题的真正最优解, 也就无法计算成员算法找到的解与最优解的距离, 从而无法判断是否是近似最优解, 最终导致无法测量算法的运行时间. 但是, 这也不会影响式(2)的成立. 进一步地, $P$在问题样例集合$I$上的性能是$P$在$I$中各个问题样例上的性能的平均值:

$$\mathrm{m}(P,I)=\frac{1}{|I|}\sum_{z\in I}\mathrm{m}(P,z), \tag{3}$$

式中, $|I|$表示集合$I$的势.

## 2.2 PAP自动构建问题

为了将PAP构建过程自动化, 本文考虑如下的优化问题: 从给定算法配置空间$\Theta$中选择$k$个成员算法组成$P$, 以使得后者在给定问题样例集合$I$上的性能达到最优.

其中, $I$被称作训练集, 其应当充分代表$P$的目标使用场景. 换言之, 如果$P$在$I$上性能良好, 那么可以推断$P$在实际使用时也具有良好的性能. 算法配置空间$\Theta$由一组参数化算法$B_1,B_2,...,B_c$所定义, 方便起见, 我们称这类算法为基础算法. 其中每个基础算法$B_i$都有若干参数, 参数的取值控制着$B_i$方方面面的行为, 进而对$B_i$的性能有着巨大的影响. 因此, 当$B_i$的参数取不同值时, 可以认为得到了不同的算法; 在此基础上, 对$B_i$的参数取遍所有可能的值, 最终得到的所有独一无二的算法所组成的集合, 就是$B_i$的算法配置空间, 记为$\Theta_i$. 例如, 假设$B_1$有两个参数$\alpha$和$\beta$, 其各有两种取值, $\alpha\in\{0,1\}$, $\beta\in\{0,1\}$; 那么$\Theta_1$一共包含4个不同的算法, 即 $\Theta_1=\{(\alpha:0,\beta:0),(\alpha:0,\beta:1),(\alpha:1,\beta:0),(\alpha:1,\beta:1)\}$, 其中$\alpha:0$表示参数$\alpha$的取值为0. 如图1所示, 多个基础算法$B_1,B_2,...,B_c$所定义的算法配置空间$\Theta=\Theta_1\cup\Theta_2\cup...\cup\Theta_c$. 例如, 假设现在除了上面例子中的$B_1$, 还考虑另一个基础算法$B_2$, 其有一个参数$\gamma\in\{0,1\}$, 因此$\Theta_2=\{(\gamma:0),(\gamma:1)\}$, 最终$\Theta=\Theta_1\cup\Theta_2$.

如前所述, $P$中每个成员算法$\theta_i$都是从$\Theta$中选择得到, 因此$\theta_i$的搜索空间大小为$|\Theta|$. 此外, 由等式(2)可知, 假设$P$已经包含成员算法$\theta_i$, 那么再将同样的算法$\theta_i$加入到$P$则不会带来任何性能增益. 因此, 在为$P$寻找新成员算法时, 可以简单地先将$P$已包含的成员算法排除掉, 这可以保证最终得到的$P$中各个成员算法互不相同. 实际上, $P$可以看作$\Theta$的子集, 那么$P$的整体搜索空间大小为$\prod_{i=1}^{k}(|\Theta|-i+1)=O(|\Theta|^k)$. 综上所述, PAP自动构建问题的形式化定义如下:

$$P^*=\mathrm{argmax}_{P\subseteq\Theta\text{且}|P|=k}\mathrm{m}(P,I), \tag{4}$$

式中, $P^*$表示最优PAP.

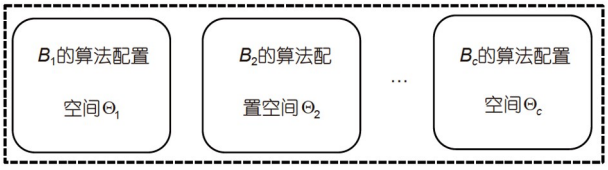


**图 1** 基础算法$B_1$, $B_2$, ..., $B_c$所定义的算法配置空间$\Theta$

**Figure 1** Algorithm configuration space $\Theta$ defined by base algorithms $B_1$, $B_2$, ..., $B_c$.

## 3 PAP智能汇聚构造方法

### 3.1 AutoPAP框架

本文提出了一种基于演化优化的方法AutoPAP来解决上文中的PAP自动构建问题. 演化算法是一类具有广泛适用性的全局优化方法[13], 其通过将种群进行交叉(crossover)、变异(mutation)等操作, 并在一定的控制参数(如演化代数)作用下, 于解空间中进行搜索. 值得注意的是, 传统演化算法演化的每个个体通常都是待求解问题的完整解; 而在AutoPAP中, 种群(而非个体)才是完整解, 对应着PAP, 种群中的个体则对应着PAP的成员算法. 也正是基于此, AutoPAP和传统演化算法有一个明显区别: 对于后者, 在搜索过程中种群规模通常保持不变; 而在AutoPAP中, 种群(PAP)规模是逐渐增大的, 这可以保证PAP的性能不会出现退化(详细分析请见3.3节引理1). AutoPAP的具体步骤如下.

**输入:** 算法配置空间$\Theta$, 训练集$I$, 性能指标m, PAP规模$k$, 每一代生成的候选算法数量$n$.

**输出:** 最终种群$P=\{\theta_1,\theta_2,...,\theta_k\}$.

**步骤1:** 初始化当前迭代计数变量$i\leftarrow 1$; 初始化种群$P\leftarrow\varnothing$.

**步骤2:** 使用变异算子在$\Theta$中搜索生成$n$个候选算法$\theta_1',\theta_2',...,\theta_n'$.

**步骤3:** 将$\theta_1',\theta_2',...,\theta_n'$在训练集$I$上进行测试, 找到对当前$P$具有最大性能增益的算法.

$$\theta_i=\mathrm{argmax}_{j\in\{1,...,n\}}\mathrm{m}(P\cup\theta_j',I). \tag{5}$$

**步骤4:** 将$\theta_i$加入到PAP中, $P\leftarrow P\cup\theta_i$.

**步骤5:** 如果$i=k$, 算法结束并返回$P$, 否则$i\leftarrow i+1$, 并重复步骤2~5.

可以看到, AutoPAP一共有$k$次迭代, 遵循$(n+1)$演化优化框架. 在每一次迭代中, AutoPAP使用变异算子从算法配置空间$\Theta$中搜索得到$n$个候选算法, 然后保留其中对当前PAP性能提升最大的那个算法加入到$P$中. 值得注意的是, 该算法有可能无法给$P$带来性能增益, 即$\mathrm{m}(P\cup\theta_i)=\mathrm{m}(P)$, 这种情况下$\theta_i$仍然会被加入到$P$中. 这样做的原因主要有两点. 首先, 加入$\theta_i$绝对不会降低$P$的性能(请见3.3节引理1); 其次, 虽然在训练集上$\theta_i$没有给$P$带来性能增益, 但在训练集没有覆盖到的问题样例上(如实际使用场景中出现的问题样例), $\theta_i$有可能会给$P$带来提升.

随着AutoPAP迭代次数增加, $P$的规模逐步增大. 显然, 变异算子在$\Theta$中的搜索效率将在很大程度上决定$P$的最终性能. 下一节将会详细介绍AutoPAP中的变异算子. AutoPAP的计算代价主要由两部分组成. 第一部分是在步骤2中使用变异算子生成候选算法的计算代价. 令$t_c$表示生成一个算法的时间, 那么这一部分的CPU时间为$nt_c$. 第二部分则是在步骤3中对候选算法进行测试的计算代价. 令测试一个算法的时间为$t_v$, 那么这一部分的CPU时间为$nt_v$. 由此, AutoPAP一次迭代所消耗的CPU时间为$n(t_c+t_v)$, 总CPU时间则为$kn(t_c+t_v)$.

### 3.2 变异算子

如2.2节所述, $\Theta$是一组参数化算法(称作基础算法)$B_1,B_2,...,B_c$的联合配置空间, 即 $\Theta=\Theta_1\cup\Theta_2\cup...\cup\Theta_c$且$|\Theta|=\sum_{i=1}^{c}|\Theta_i|$. 换言之, $|\Theta|$将会随着基础算法的数量线性增长. 当基础算法本身的配置空间较大, 或者基础算法数量较多时, $|\Theta|$会变得非常巨大, 在这种情况下直接对$\Theta$进行搜索效率很低. 其次, $\Theta$可能会涉及不同类型的参数, 包括浮点型(如演化算法的变异速率)、整数型(如演化算法的种群规模)以及范畴型(如演化算法的种群初始化方式), 这要求变异算子能同时处理各种类型的决策变量.

本文使用了基于模型的顺序采样技术(sequential model-based algorithm configuration, SMAC[14,15])来实现变异算子. 具体而言, 变异算子调用SMAC从$\Theta$中寻找一个能够最大程度提高当前PAP性能的算法, 作为最终生成的候选算法. 在具体搜索过程中, SMAC会建立回归模型(regression model)以预测给定算法配置的性能, 然后在该模型基础之上最大化特定的获取函数(acquisition function)来决定下一个算法配置空间中的采样点. 因为使用了随机森林来建立回归模型, SMAC可以处理浮点型、整数型、范畴性甚至是序数型(ordinal)变量. 感兴趣的读者可以查阅文献[14]以进一步了解SMAC. 本文使用了SMAC的开源版本v3, 可在https://github.com/automl/SMAC3获取到.

为了解决$|\Theta|$过大而难以高效搜索的问题, 本文采用了分而治之的策略. 具体而言, 考虑到不同基础算

法$B_1, B_2, ..., B_c$的配置空间互相没有交集, 因此可以将$\Theta$自然地分解为$c$个子空间$\Theta_1, \Theta_2, ..., \Theta_c$, 并在每个子空间内部使用SMAC进行搜索. 当$c \geq 2$时, 子空间远小于原空间, 这可以大幅降低搜索难度. 进一步地, 在$c$个子空间内部分别搜索最终将会得到$c$个新算法, 而AutoPAP在每一代都要生成$n$个新算法. 因此, 可以将这$n$个新算法均匀地分配到不同子空间内部的SMAC搜索进程上. 例如, 假设$n = 10$, $c = 2$, 那么$\theta_1', \theta_3', \theta_5', \theta_7', \theta_9'$由SMAC在$\Theta_2$中搜索得到, $\theta_2', \theta_4', \theta_6', \theta_8', \theta_{10}'$则由SMAC在$\Theta_1$中搜索得到.

综上所述, AutoPAP中变异算子的具体步骤如下.

**输入:** 算法配置空间$\Theta = \Theta_1 \cup \Theta_2 \cup ... \cup \Theta_c$, 训练集$I$, 性能指标m, 每一代生成的候选算法数量$n$, 当前$P$.

**输出:** $n$个候选算法$\theta_1', \theta_2', ..., \theta_n'$.

**步骤1:** 初始化算法标记变量$i \leftarrow 1$.

**步骤2:** $j \leftarrow (i \bmod c) + 1$.

**步骤3:** 使用SMAC在$\Theta_j$中搜索

$\theta_i' = \operatorname{argmax}_{\theta \in \Theta_j} \mathrm{m}(P \cup \{\theta\}, I)$.

**步骤4:** 如果$i = n$, 算法结束并返回$\theta_1', \theta_2', ..., \theta_n'$, 否则$i \leftarrow i + 1$, 并重复步骤2~4.

在变异算子的步骤2中, mod表示取模运算. 在步骤3中, SMAC的优化目标是$\max_{\theta \in \Theta_j} \mathrm{m}(P \cup \{\theta\}, I)$. 当运行时间$t_c$趋向于无穷时, SMAC以概率1找到最优算法; 在实际中, $t_c$不可能为无穷大, 因此SMAC最终返回的往往是近似最优算法. 值得注意的是, 只有当$n$不小于$c$时才能保证每个子空间都会被变异算子搜索至少一次, 因此在使用AutoPAP时需设置$n \geq c$.

### 3.3 理论分析

本节从理论上分析AutoPAP在式(4)所定义的PAP自动构造问题上所能达到的近似比. 简便起见, 以$f(P)$表示$\mathrm{m}(P, I)$, 首先证明如下引理1和引理2.

**引理1.** 式(4)中的目标函数具有单调性, 即对于任意$A \subseteq B \subseteq \Theta$, 满足$f(A) \leq f(B)$.

**证明:** 根据等式(2), 对于任意问题样例$z$, 以下不等式成立.

$$\mathrm{m}(B, z) = \max\{\mathrm{m}(A, z), \mathrm{m}(B \setminus A, z)\} \geq \mathrm{m}(A, z).$$

进一步地, 结合式(3), 以下不等式成立.

$$\begin{aligned} f(B) - f(A) &= \mathrm{m}(B, I) - \mathrm{m}(A, I) \\ &= \frac{1}{|I|} \sum_{z \in I} [\mathrm{m}(B, z) - \mathrm{m}(A, z)] \\ &\geq 0. \end{aligned}$$

证明完毕.

**引理2.** 式(4)中的目标函数$f$具有子模性, 即对于任意$A \subseteq B \subseteq \Theta$和任意$\theta \in \Theta \setminus B$, 满足

$$f(A \cup \{\theta\}) - f(A) \geq f(B \cup \{\theta\}) - f(B). \tag{6}$$

**证明:** 根据等式(2), 对于任意问题样例$z$, 以下等式成立.

$$\begin{aligned} &\mathrm{m}(A \cup \{\theta\}, z) = \max\{\mathrm{m}(A, z), \mathrm{m}(\theta, z)\}, \\ &\mathrm{m}(B \cup \{\theta\}, z) = \max\{\mathrm{m}(A, z), \mathrm{m}(B \setminus A, \theta), \mathrm{m}(\theta, z)\}. \end{aligned}$$

考虑以下三种不同情况.

情况(1): $\mathrm{m}(\theta, z) < \mathrm{m}(A, z)$, 那么如下等式成立.

$$\begin{aligned} &\mathrm{m}(A \cup \{\theta\}, z) = \mathrm{m}(A, z), \\ &\mathrm{m}(B \cup \{\theta\}, z) = \mathrm{m}(B, z). \end{aligned}$$

由此得到

$$f(A \cup \{\theta\}) - f(A) = f(B \cup \{\theta\}) - f(B) = 0.$$

情况(2): $\mathrm{m}(B, z) \geq \mathrm{m}(\theta, z) \geq \mathrm{m}(A, z)$, 那么如下等式成立.

$$\begin{aligned} &\mathrm{m}(A \cup \{\theta\}, z) = \mathrm{m}(\theta, z), \\ &\mathrm{m}(B \cup \{\theta\}, z) = \mathrm{m}(B, z). \end{aligned}$$

由此得到

$$f(A \cup \{\theta\}) - f(A) \geq 0 = f(B \cup \{\theta\}) - f(B).$$

情况(3): $\mathrm{m}(B, z) < \mathrm{m}(\theta, z)$, 那么如下等式成立.

$$\begin{aligned} &\mathrm{m}(A \cup \{\theta\}, z) = \mathrm{m}(\theta, z), \\ &\mathrm{m}(B \cup \{\theta\}, z) = \mathrm{m}(\theta, z). \end{aligned}$$

进一步地, 可以得到如下不等式.

$$\begin{aligned} \mathrm{m}(A \cup \{\theta\}, z) - \mathrm{m}(A, z) &= \mathrm{m}(\theta, z) - \mathrm{m}(A, z) \\ &\geq \mathrm{m}(\theta, z) - \mathrm{m}(B, z) \\ &= \mathrm{m}(B \cup \{\theta\}, z) - \mathrm{m}(B, z). \end{aligned}$$

这表明$f(A \cup \{\theta\}) - f(A) \geq f(B \cup \{\theta\}) - f(B)$.

证明完毕.

直观上, 引理1意味着随着PAP的规模增大, 其性能单调非递减. 引理2则意味着对于PAP来说, 添加成员算法所带来的性能增益具有如下的边际递减效应:

将成员算法$\theta$添加到$A$的性能收益总是大于或等于将$\theta$添加到$A$的超集$B$的性能收益. 简便起见, 将所谓的性能收益记为

$$\boldsymbol{\Delta}(\theta \mid A) = f(A \cup \{\theta\}) - f(A). \tag{7}$$

下面的定理1表明了AutoPAP在理论上可以达到$(1-1/e)$近似最优构造效果. 为了证明定理1, 定义$P_i$为AutoPAP运行过程中第$i$次迭代结束时的PAP, 即$P_i = \{\theta_1, ..., \theta_i\}$, 并令$P^* = \{\theta_1^*, ..., \theta_k^*\}$表示最优PAP, 即$P^* = \text{argmax}_{P \subseteq \Theta \text{且} |P|=k} f(P)$.

**定理1.** 当$t_c \to \infty$, 且$n \geq c$时, 对于任意正整数$i$和$k$, 满足$f(P_i) \geq (1-e^{-i/k}) f(P^*)$. 特别地, 当$i=k$时, $f(P_k) \geq (1-1/e) f(P^*)$.

**证明:** 如前所述, 在AutoPAP的每次迭代中, $t_c \to \infty$可以保证每个子配置空间中对当前PAP性能提升最大的算法被找到, $n \geq c$则可以保证每个子配置空间都会被变异算子搜索至少一次. 综上, AutoPAP在第$i$次迭代最终找到的算法$\theta_i$满足

$$\theta_i = \text{argmax}_{\theta \in \Theta} \boldsymbol{\Delta}(\theta \mid P_{i-1}).$$

因此, 如下不等式成立.

$$\begin{aligned} f(P^*) &\leq f(P^* \cup P_i) \quad \text{引理1} \\ &= f(P_i) + \sum_{j=1}^{k} \boldsymbol{\Delta}\left(\theta_j^* \mid P_i \cup \left\{\theta_1^*, \theta_2^*, ..., \theta_{j-1}^*\right\}\right) \\ &\leq f(P_i) + \sum_{\theta \in P^*} \boldsymbol{\Delta}(\theta \mid P_i) \quad \text{引理2} \\ &\leq f(P_i) + \sum_{\theta \in P^*} \boldsymbol{\Delta}(\theta_{i+1} \mid P_i) \\ &= f(P_i) + k\,\boldsymbol{\Delta}(\theta_{i+1} \mid P_i). \end{aligned}$$

重新整理上式, 得到以下不等式.

$$\boldsymbol{\Delta}(\theta_{i+1} \mid P_i) \geq \frac{1}{k}(f(P^*) - f(P_i)). \tag{8}$$

定义$\boldsymbol{\delta}_i = f(P^*) - f(P_i)$, 意味着如下等式成立.

$$\boldsymbol{\delta}_i - \boldsymbol{\delta}_{i+1} = f(P_{i+1}) - f(P_i) = \boldsymbol{\Delta}(\theta_{i+1} \mid P_i). \tag{9}$$

将式(9)代入不等式(8), 得到以下不等式.

$$\boldsymbol{\delta}_i - \boldsymbol{\delta}_{i+1} \geq \frac{\boldsymbol{\delta}_i}{k}. \tag{10}$$

将不等式(10)重新整理, 并递归应用, 得到下式.

$$\begin{aligned} \boldsymbol{\delta}_i &\leq \left(1 - \frac{1}{k}\right)^i \boldsymbol{\delta}_0 \\ &\leq \left(1 - \frac{1}{k}\right)^i [f(P^*) - f(\varnothing)] \\ &\leq \left(1 - \frac{1}{k}\right)^i f(P^*). \end{aligned}$$

基于$\left(1 - \frac{1}{k}\right) \leq \mathrm{e}^{-1/k}$, 得到$\boldsymbol{\delta}_i \leq \mathrm{e}^{-i/k} f(P^*)$. 因为$\boldsymbol{\delta}_i = f(P^*) - f(P_i)$, 那么下式成立.

$$f(P_i) \geq (1 - \mathrm{e}^{-i/k}) f(P^*).$$

证明完毕.

## 4 实验验证

本文以TSP问题为例, 验证AutoPAP的有效性. TSP[16]是著名的NP难问题, 其描述如下: 给定若干城市, 寻找一条遍历所有城市的最短路径. TSP在物流、通讯、电路设计等领域有着广泛应用[17], 多年来一直吸引着众多研究人员对其求解器进行持续改进. 目前, 学术界公认的最佳TSP求解器是采用Lin-Kernighan启发式[17]的LKH[11,18]以及使用边组装交叉算子的EAX[12,19]. 本文将使用AutoPAP针对TSP问题构造PAP (记为TSP_PAP), 然后将TSP_PAP和这些求解器进行比较.

### 4.1 问题样例集

为了尽可能全面地测试求解器的性能, 本文从TSP基准集网站http://www.math.uwaterloo.ca/tsp收集了4种类型共128个规模1000~30000的TSP问题样例. 这四种类型分别是:

(a) TSPlib[20]来自实际应用, 共31个样例;

(b) National TSP, 从各国地图抽象得到, 共19个样例;

(c) VLSI TSP, 来自超大规模集成电路设计场景, 共72个样例;

(d) DIMACS Test Set[16], 曾在DIMACS TSP求解挑战中作为基准测试集, 共6个样例.

表1总结了这些问题样例在各规模区间和各问题

**表 1** 实验所采用的TSP问题样例在各问题规模区间上的分布情况

**Table 1** Distribution of the TSP problem instances over each problem size interval

| 规模区间 | [1000,5000] | [5001,10000] | [10001,15000] | [15001,20000] | [20001,25000] | [25000,30000] | 总计 |
|---|---|---|---|---|---|---|---|
| TSPlib | 23 | 3 | 3 | 2 | 0 | 0 | 31 |
| National | 4 | 11 | 2 | 0 | 2 | 0 | 19 |
| VLSI | 48 | 7 | 3 | 4 | 4 | 6 | 72 |
| DIMACS | 0 | 0 | 6 | 0 | 0 | 0 | 6 |
| 总计 | 75 | 21 | 14 | 6 | 6 | 6 | 128 |

类型上的分布情况. 一般而言, 当TSP问题规模超过1000时, 就被认为是中等规模TSP, 而超过5000时则被认为是大规模TSP. 可以看到本文所采用的TSP样例全面覆盖了小、中、大三种规模. 此外, 这些问题样例也覆盖了多种TSP距离类型, 包括定义在二维、三维、四维欧式距离上的EUC_2D, EUC_3D, EUC_4D类型, 直接以距离矩阵给出的MATRIX类型, 以及地理距离类型GEO, 感兴趣的读者可以查询文献[20]获取这些距离的详细定义. 值得注意的是, TSP_PAP适用的TSP距离类型是由其基础算法LKH和EAX决定的(请见4.3节). 因此, 和LKH以及EAX一样, TSP_PAP对于以上所有距离类型都适用. 进一步地, 本文收集了这些TSP样例的最优解作为评判标准以衡量求解器找到的解的好坏. 由于部分TSP样例的最优解目前仍然未知, 对于这些样例本文使用当前已知最好的解(best known solution)作为最优解.

### 4.2 性能指标

本文使用了两种指标来衡量求解器的性能. 其中, 带10倍惩罚的平均运行时间(penalized average runtime-10, PAR10)衡量了求解器找到TSP样例最优解所需要的平均时间(即求解效率), 值越小越好. 在实际操作中, 被测试的求解器将会被限定最长运行时间$T_{\max}$=3600 s. 如果在运行了$t \le T_{\max}$时求解器成功找到了问题样例的最优解, 那么此次运行视为"成功", 且时间记录为$t$; 否则此次运行视为"超时", 时间记录为$10T_{\max}$. 最终, 将求解器在所有样例上所需运行时间的平均值记为PAR10.

此外, 本文使用了平均误差率(average deviation ratio, ADR)来衡量求解器找到的解的质量(即求解效果), 值越小越好. 在实际操作中, 被测试的求解器将会被限定最长运行时间$T_{\max}$=3600 s, 当运行$T_{\max}$后, 假定求解器返回的解的代价为$cost$, 而问题样例的最优解的代价为$cost^*$, 则计算

$$\mathrm{DR} = \frac{cost - cost^*}{cost^*}. \tag{11}$$

最终, 将求解器在所有样例上获得的DR的平均值记为ADR.

### 4.3 构造TSP_PAP

如前所述, AutoPAP的输入为训练集$I$和算法配置空间$\Theta$, 后者由若干基础算法定义. 本文从128个TSP样例中随机抽取了50个样例构成训练集$I$, 并选择了LKH[11](版本2.0.9)和EAX[12](版本1.0)作为AutoPAP的基础算法. LKH一共有29个参数, EAX一共有8个参数.

考虑到如今4核CPU已经相当常见, 本文将AutoPAP的参数$k$(成员算法数量)设置为4, $n$(每一代生成候选算法的数量)设置为10. AutoPAP优化的性能指标m为PAR10, 生成单个算法的时间$t_c$为24 h, 测试单个算法的时间$t_v$为6 h. 如3.1节所述, AutoPAP消耗的总CPU时间为$kn(t_c+t_v)$, 因此本实验中AutoPAP一共使用了1200 CPU小时(共50 CPU天)构造TSP_PAP. 另一方面, 由于AutoPAP的每一次迭代中变异算子生成10个新算法并测试的过程是相互独立的, 因此可以并行运行, 那么在CPU核数大于等于10的计算平台上, TSP_PAP的构造时长可以缩短为$k(t_c+t_v)$ = 5天. 实验中所用计算平台为Intel Xeon CPU E5-2699A v4@2.40 GHz, 22核心, 55 MB高速缓存.

### 4.4 基线求解器

本文考虑了如下TSP上公认性能最好的两个求解器作为基线求解器:

(a) LKH[11], 本文所考虑的基础求解器之一, 使用默认参数配置;

(b) EAX[12], 本文所考虑的基础求解器之一, 使用默认参数配置.

考虑到LKH和EAX的性能会受其参数配置影响, 本文进一步使用SMAC对LKH和EAX分别进行自动算法配置, 其中训练集和总CPU时间与构造TSP_PAP时保持一致:

(c) LKH-TUNED, 自动算法配置得到的LKH;

(d) EAX-TUNED, 自动算法配置得到的EAX.

考虑到TSP_PAP包含了4个算法, 而以上各求解器实际上仅为一个算法, 因此我们人为地将以上4个算法进一步组成PAP, 得到:

(e) EAX_LKH, 由以上4个算法组成.

此外, 为了验证AutoPAP中分而治之策略的有效性, 还得到了如下PAP:

(f) TSP_PAP*, 在构造该PAP过程中使用变异算子直接搜索整个算法配置空间, 而非子配置空间.

### 4.5 实验结果与分析

在每个TSP样例上, 每个求解器被重复运行3次(每次使用不同的随机种子), 3次运行结果的中位数作为该求解器在这个问题样例上的结果, 在此基础上计算出每个求解器在整个TSP样例集上的超时数(超时意味着运行3600 s仍未找到问题样例的最优解), PAR10和ADR. 表2展示了各求解器的测试结果, 其中每个性能指标上的最佳结果以"_"标记.

首先, TSP_PAP在所有求解器中获得了最少的超时数、最短的运行时间和最小的平均误差率, 这意味着无论是在求解效率还是在求解效果上, TSP_PAP都显著好于其余基线求解器. 图2直观对比了TSP_PAP和LKH以及EAX. 可以看到, 相对于LKH和EAX, TSP_PAP可以将平均求解时间大幅缩短至少45.71%, 而在平均误差率上的降低幅度更为巨大, 达到了至少87.50%. 注意以上结果是在已经对LKH和EAX进行了算法配置的基础之上达成的, 这说明TSP_PAP已经成为了TSP上综合性能最强的求解器, 也验证了AutoPAP作为PAP构造方法的有效性.

相比于TSP_PAP*, TSP_PAP将超时数减少了28.57%, 将PAR10缩短了20.49%, 将ADR降低了94.87%, 这说明AutoPAP采取的分而治之策略确实能够让变异算子更加高效地搜索算法配置空间, 从而构

**表 2** 各求解器在TSP样例集上的超时数量(#Timeous), PAR10以及ADR

**Table 2** The number of timeouts (#Timeouts), PAR10 and ADR obtained by each TSP solver

| | TSP_PAP | LKH | EAX | LKH-TUNED | EAX-TUNED | EAX_LKH | TSP_PAP* |
|---|---|---|---|---|---|---|---|
| #Timeouts | 7 | 50 | 16 | 43 | 14 | 9 | 9 |
| PAR10(s) | 2369.38 | 14242.85 | 4928.88 | 12319.73 | 4364.25 | 2921.51 | 2980.09 |
| ADR(‰) | 0.02 | 0.56 | 0.24 | 0.51 | 0.16 | 0.12 | 0.39 |

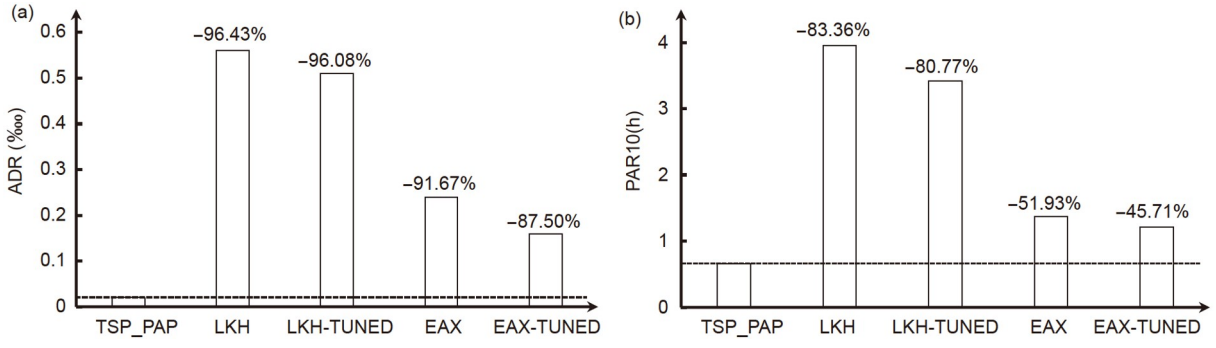


**图 2** TSP_PAP以及LKH, EAX的平均误差率(ADR) (a)和平均求解时间(PAR10) (b), 百分数表示TSP_PAP相比于对应求解器的性能提升幅度

**Figure 2** The ADR (a) and PAR10 (b) obtained by TSP_PAP, LKH and EAX. The percentages indicate the performance improvement achieved by TSP_PAP, compared to the corresponding solvers.

造出性能更强的PAP. 此外, TSP_PAP相比于EAX_LKH也取得了类似幅度的性能提升, 考虑到后者是人为将已有TSP求解器组合而成, 这表明了AutoPAP所遵循的“PAP自动构造”的方法论相对于“手工构造PAP”有性能优势.

虽然EAX和LKH都是目前公认最佳的TSP求解器, 但在表2中前者的表现比后者要好得多, 这与近期文献[21]结果相符. 此外, 经过自动算法配置之后, EAX和LKH的性能都有了小幅提升, 这表明参数取值确实会影响求解器的性能.

我们进一步分析了在不同问题规模区间上TSP_PAP的性能. 对于规模不大于5000的中小TSP样例, TSP_PAP的PAR10为40.75 s, 且ADR为0(即全部都找到了最优解). 相比较而言, EAX_LKH在中小规模TSP样例上的ADR为0.03‰, PAR10则为109.57 s, 后者是TSP_PAP所需时间的2.7倍. 随着TSP规模变大, 问题的解空间规模指数级增长, TSP_PAP所需求解时间也显著增长. 在规模大于5000的TSP样例上, TSP_PAP的PAR10为5664.61 s, 而EAX_LKH则为6900.67 s. 后者为前者的1.2倍.

最后, 可以看到表2中所有PAP型求解器(TSP_PAP, EAX_LKH以及TSP_PAP*)的表现都远好于非PAP型求解器, 这表明了PAP这种求解器形式所具有的巨大优越性.

### 4.6 TSP_PAP性能变化趋势

图3展示了在AutoPAP运行过程中每一代结束时的PAP($P_1, P_2, P_3, P_4$)在TSP样例集上的测试结果. 注意图中的横坐标以AutoPAP消耗的CPU时间给出. 可以看到, 随着迭代次数增多, PAP的性能单调提升, 这符合引理1. 此外, AutoPAP消耗300个CPU小时生成的$P_1$(仅包含一个成员算法)和EAX-TUNED性能相近, 而消耗900个CPU小时生成的$P_3$(包含3个成员算法)的性能则超过了同为PAP的EAX_LKH. AutoPAP消耗了1200个CPU小时生成了$P_4$, 从图3中可以看出, 此时PAR10下降趋势仍未停止. 可以预想的是, 随着AutoPAP迭代次数的进一步增加, TSP_PAP的性能将进一步提升.

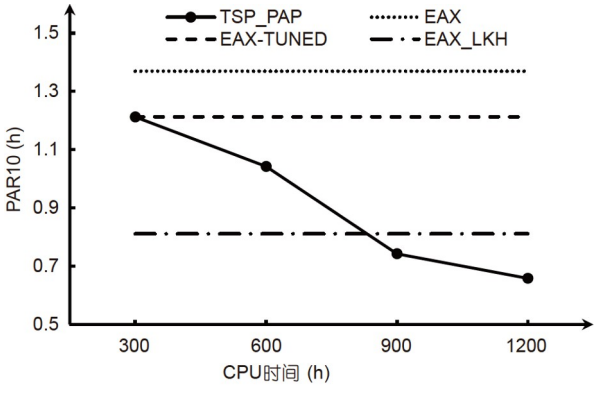


**图 3** 随着AutoPAP所消耗的CPU时间增多, TSP_PAP的性能(PAR10)变化趋势

**Figure 3** The performance (in terms of PAR10) of TSP_PAP against the CPU time consumed by AutoPAP.

## 5 总结

本文提出了一种基于演化优化的PAP智能汇聚构造方法AutoPAP, 其可以基于给定算法配置空间和训练集自动构造出高性能PAP. AutoPAP在理论上可以达到$(1-1/e)$近似最优构造效果, 其针对TSP问题所构造的TSP_PAP的性能大幅超过了EAX以及LKH, 成为了综合性能最强的TSP求解器.

TSP_PAP的强悍性能充分展示了AutoPAP的巨大应用价值. 事实上, 只需提供相应算法配置空间和训练集, AutoPAP可以被用于针对几乎任何问题类构建PAP. 本文下一步工作便是将AutoPAP应用到其他问题领域如车辆路径规划问题[22]、云服务组合[23]以及群体机器人控制[24].

另一个重要的研究方向是对AutoPAP进行改进. 当前AutoPAP的局限性主要体现在两方面. 首先, 由引理2可知, 随着PAP成员算法增多, 对PAP添加更多成员算法所带来的性能增益呈现边际递减效应, 这表明当前AutoPAP有造成“计算资源浪费”的风险, 即对PAP性能增益有限的成员算法和对PAP性能增益明显的成员算法使用了同样的计算资源. 造成这一现象的根本原因在于AutoPAP当前仅能添加成员算法, 而不涉及删除、替换等操作, 未来可将这些操作引入AutoPAP以提高灵活性. AutoPAP的第二个局限性在于当前变异算子的分而治之策略仅对基础算法数量大于2的情况有效. 当基础算法数量为1时, 可尝试将该算法的配置空间进行分解. 具体而言, 可首先分析找出对该算法行为有决定性影响的参数, 然后将该参数的取值范围均分为不相交的若干集合, 从而得到分解方案. 在此基础之上, 即可按照当前策略在各子空间中并行搜索

以达到加速效果. 最后, 还可以借助演化计算中较为成熟的多种群技术[25]、负相关搜索技术[26]等, 进一步提高AutoPAP中变异算子对算法配置空间的搜索效率.

## 参考文献


1 Asanovic K, Bodik R, Demmel J, et al. A view of the parallel computing landscape. Commun ACM, 2009, 52: 56–67
2 Biere A, Fazekas K, Fleury M, et al. CaDiCaL, Kissat, paracooba, plingeling and treengeling entering the SAT competition 2020. In: Proceedings of SAT Competition 2020: Solver and Benchmark Descriptions. Helsinki: University of Helsinki, 2020. 50–53
3 Gottlob G, Okulmus C, Pichler R. Fast and parallel decomposition of constraint satisfaction problems. In: Proceedings of the 29th International Joint Conference on Artificial Intelligence. AAAI Press, 2020. 1155–1162
4 Gebser M, Kaufmann B, Neumann A, et al. Clasp: A conflict-driven answer set solver. In: Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning. Tempe, 2007. 260–265
5 Ralphs T K, Shinano Y, Berthold T, et al. Parallel solvers for mixed integer linear optimization. In: Hamadi Y, Sais L, eds. Handbook of Parallel Constraint Reasoning. Cham: Springer, 2018. 283–336
6 Tang K, Peng F, Chen G, et al. Population-based algorithm portfolios with automated constituent algorithms selection. Inf Sci, 2014, 279: 94–104
7 Hamadi Y, Wintersteiger C M. Seven challenges in parallel SAT solving. AI Mag, 2013, 34: 99–106
8 Liu S C, Tang K, Yao X. Automatic construction of parallel portfolios via explicit instance grouping. AAAI, 2019, 33: 1560–1567
9 Liu S C, Tang K, Yao X. Generative adversarial construction of parallel portfolios. IEEE Trans Cybern, 2020, 1–12
10 Tang K, Liu S C, Yang P, et al. Few-shots parallel algorithm portfolio construction via co-evolution. IEEE Trans Evol Computat, 2021, 25: 595–607
11 Helsgaun K. General k-opt submoves for the Lin-Kernighan TSP heuristic. Math Prog Comp, 2009, 1: 119–163
12 Nagata Y, Kobayashi S. A powerful genetic algorithm using edge assembly crossover for the traveling salesman problem. INFORMS J Computing, 2013, 25: 346–363
13 DeJong K. Evolutionary Computation: A Unified Approach. Cambridge: MIT Press, 2006
14 Hutter F, Hoos H H, Leyton-Brown K. Sequential model-based optimization for general algorithm configuration. In: Proceedings of the 5th International Conference on Learning and Intelligent Optimization. Rome, 2011. 507–523
15 Liu S C, Tang K, Yao X. On performance estimation in automatic algorithm configuration. In: Proceedings of the 34th AAAI Conference on Artificial Intelligence. New York, 2020. 2384–2391
16 Gutin G, Punnen A P. The Traveling Salesman Problem and Its Variations. New York: Springer, 2006
17 Lin S, Kernighan B W. An effective heuristic algorithm for the traveling-salesman problem. Oper Res, 1973, 21: 498–516
18 Helsgaun K. An effective implementation of the Lin-Kernighan traveling salesman heuristic. Eur J Oper Res, 2000, 126: 106–130
19 Nagata Y, Kobayashi S. Edge assembly crossover: A high-power genetic algorithm for the travelling salesman problem. In: Proceedings of the 7th International Conference on Genetic Algorithms. East Lansing, 1997. 450–457
20 Reinelt G. TSPLIB—A traveling salesman problem library. ORSA J Comput, 1991, 3: 376–384
21 Nagata Y. High-order entropy-based population diversity measures in the traveling salesman problem. Evol Comput, 2020, 28: 595–619
22 Liu S C, Tang K, Yao X. Memetic search for vehicle routing with simultaneous pickup-delivery and time windows. Swarm Evol Comput, 2021, 66: 100927
23 Liu S C, Wei Y F, Tang K, et al. QoS-aware long-term based service composition in cloud computing. In: Proceedings of the 2015 IEEE Congress on Evolutionary Computation. Sendai: IEEE, 2015. 3362–3369
24 Wang Y, Ma L, He Y M, et al. A multi-objective optimization method for intelligent swarm robotic control model with changeable parameters (in Chinese). Sci Sin Tech, 2020, 50: 526–537 [王原, 马力, 王凌, 等. 智能机器人可变参数群体控制模型的多目标优化方法. 中国科学: 技术科学, 2020, 50: 526–537]
25 Ma H, Shen S, Yu M, et al. Multi-population techniques in nature inspired optimization algorithms: A comprehensive survey. Swarm Evol Comput, 2019, 44: 365–387
26 Tang K, Yang P, Yao X. Negatively correlated search. IEEE J Sel Areas Commun, 2016, 34: 542–550

# Approximately optimal construction of parallel algorithm portfolios by evolutionary intelligence

LIU ShengCai[1,2], YANG Peng[1,2] & TANG Ke[1,2]

[1] *Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen 518055, China;*
[2] *Guangdong Key Laboratory of Brain-Inspired Intelligent Computation, Shenzhen 518055, China*

As a high-performance general-purpose form of parallel solvers, parallel algorithm portfolios (PAPs) have recently shown great performance for solving decision, counting, continuous, and discrete optimization problems. However, the manual construction of PAPs is laborious work, which typically requires substantial domain knowledge and human effort. To address this issue, this paper proposes AutoPAP, an evolutionary intelligence-based method for PAP construction. Overall, AutoPAP follows the ($n$+1) evolutionary framework, i.e., in each generation, $n$ candidate algorithms are generated, and the best one among them is retained in the PAP. Considering that the algorithm configuration space is typically very large, this study designs a highly effective mutation operator to improve the practical performance of AutoPAP and further theoretically proves that AutoPAP can achieve (1−1/$e$) approximation. Finally, to validate the effectiveness of AutoPAP, this study uses it to build a PAP, namely, TSP_PAP, for traveling salesman problems (TSPs). The test results show that TSP_PAP significantly outperforms state-of-the-art TSP solvers EAX and LKH in terms of efficiency (runtime) and effectiveness (solution quality). On 128 TSP instances of sizes ranging from 1000 to 30000, compared with LKH and EAX, TSP_PAP can reduce the average runtime by at least 45.71% and lower the average deviation ratio by at least 87.50%, indicating the huge potential of AutoPAP in automatic algorithm design and evolution.

**problem solver, parallel algorithm portfolios, evolutionary computation, automatic algorithm design, traveling salesman problem, combinatorial optimization**

doi: 10.1360/SST-2021-0372