多处理器系统上的具有约束关系的任务调度问题

——算法综述和复现

摘要

多处理器系统是指由多个可以互联通信的处理器组成的系统. 在多处理器系统上，并行程序需要在满足一系列约束条件的前提下执行完成由多个相对独立的小型任务组成的大型任务，这就引出了按何种顺序执行任务的问题，即任务调度的问题.

任务调度问题可以通过带权的有向无环图 (DAG) 加以建模. 有向无环图的结点代表任务，有向边代表任务之间的前后依赖关系，边权值代表通信代价. 一般的任务调度问题是 NP– 完全的问题，且不存在伪多项式时间的算法. 因此，启发式的算法是解决任务调度问题的最为常用思路.

启发式算法建立在贪心的基本思想之上，可以分为传统启发式算法和元启发式算法两类. 本问题的传统启发式算法又可以分为三类：基于列表的算法、基于聚类的算法和基于副本的算法，每一类算法解决问题的侧重点又有所不同.HEFT 算法和 CPOP 算法是经典的基于列表的算法，DSC 算法是一种基于聚类的算法，CPFD 算法是一种优秀的基于副本的算法.

元启发式算法的代表是遗传算法. 利用遗传算法解决此问题的算法最早由 1994 年的一篇论文提出. 通过复现该论文中所述的算法并进行实验，可以观察到遗传算法的强大能力. 不过，单纯的遗传算法在解决问题时仍显得较为单薄、不够可靠，尤其是对于稀疏图的表现并不一定非常良好. 因此，原论文中的实验结果也有令人怀疑的部分. 此外，该遗传算法具有不平衡性、搜索空间不完整性和无知性三个缺陷. 后续的研究主要针对第三个缺陷加以改进，而针对前两个缺陷也具有改进方案.

将传统启发式算法的部分思想与遗传算法相结合，从而去除遗传算法本身的无知性，往往可以达到非常良好的效果. 一种增强的任务调度遗传算法就是利用这个思路实现的.

关键词： 多处理器系统 任务调度问题 有向无环图 启发式搜索算法 遗传算法

目录

一、 问题介绍和数学描述

1.1 问题简介

设一个多处理器系统 (multiprocessor system) 由 m 个处理器组成，$m > 1$. 每个处理器都有自己的内存，每一对处理器通过互联网络进行独立通信. 一个并行程序 (parallel program) 需要处理一组具有许多优先级约束的任务. 其中，每一个任务都具有一个成本，表示它的执行时间. 为了能够顺利执行并行程序，每个任务必须被调度到一个给定的多处理器系统的某个处理器上. 因此，在并行程序中两个具有优先级关系的任务可能会被调度到不同的处理器上，这就导致不同处理器在并行程序执行期间可能会进行通信 (communication)，这通常会降低程序的执行速度. 可行的调度策略是多种多样的，而针对一个给定任务集合，不同调度策略所得到的的并行程序的运行时间可能会有很大的差距. 因此，给定任务集合，如何寻找优良的调度策略是一个非常有趣且关键的问题.

本文中，我们梳理了该问题自提出以来的研究发展历程，阐述其中具有重要意义的算法，并以近年来广泛应用的遗传算法为例进行复现和实验，在实验的基础上尝试对复现算法给出改进方案. 在第一部分中，我们给出了多处理器系统上具有约束关系的任务调度问题的形式化的数学定义；在第二部分中，我们梳理传统启发式算法在该问题上的应用，着重分析四个经典的算法模型；在第三部分中，我们介绍一种使用遗传算法的调度算法，并针对该算法实现了代码复现，在此基础上我们进行实验，分析了实验结果和原始算法中存在的问题和缺陷，并提出了改进方案，针对改进方案又进行了对比实验；第四部分中，我们进一步梳理近年来遗传算法在该问题中的广泛应用.

1.2 数学描述和定义

为了形式化多处理器集上的具有约束关系的任务调度问题，我们用数学符号给出相关概念和定义. 一个多处理器系统包含一个处理器集 $\mathcal{P} = \{p_1, p_2, ..., p_m\}$，其中 $p_1, p_2, ..., p_m$ 代表 m 个处理器. 它们通过一个完全的通信网络相连接, 且每个连接相互独立. 每个处理器在同一时间最多只能执行一个任务，并且不允许抢占 (preemption). 处理完一个任务后，处理器可以通过通信网络将计算结果等信息发送给其他处理器.

具有约束关系的任务集合可以表示为一个带权的的有向无环图 $\mathcal{D} = (\mathcal{T}, E, v, w)$. 图中的顶点代表任务集 $\mathcal{T} = \{t_1, t_2, ..., t_n\}$. 有向边代表任务之间的约束关系，也即是一种偏序关系. 边 $(t_{i_1}, t_{i_2}) \in E$ 代表，当任务 t_{i_1} 执行结束后，需要向任务 t_{i_2} 发送一些执行任务 t_{i_2} 所必须的信息，也即任务 t_{i_2} 必须在任务 t_{i_1} 完成后才能开始. 此时称 t_{i_2} 是 t_{i_1} 的直接后继，t_{i_1} 是 t_{i_2} 的直接前驱. 给出处理某个任务所需要的指令数. **图 1** 是有向无环图的一个简单示例，其共有 9 个顶点，代表 9 项任务，任务间的约束关系用顶点之间的有向边表示. 根据我们的定义，**图 1** 的数学表示为：$\mathcal{T} = \{t_1, t_2, ..., t_9\}$, $E = \{< t_1, t_3 >, < t_1, t_4 >, < t_2, t_3 >, < t_2, t_8 >, < t_3, t_5 >, < t_4, t_5 >, < t_4, t_6 >, < t_4, t_7 >, < t_5, t_8 >, <$

$t_6, t_9 >, < t_7, t_9 >, < t_8, t_9 >\}$, $\mathcal{D} = (\mathcal{T}, E)$.

有向无环图的每条有向边 e 都带有权值 $w(e)$，表示任务间通信所需要花费的时间代价，称为通信代价. 当任务 t_i 是任务 t_j 的直接前驱，且任务 t_i 和 t_j 不在同一个处理器上处理时，系统需要花费 $w(< t_i, t_j >)$ 的时间进行数据通信，即任务 t_i 完成后再经过 $w(< t_i, t_j >)$ 的时间，任务 t_j 才能够开始进行. 当任务 t_i 和 t_j 在同一个处理器上处理时，则不会产生通信代价，因为任务 t_j 所需要的信息可以在 t_i 完成后立即得到. 对于一个轻量级的系统，通信时间相较于任务处理时间往往可以忽略，因此我们也可以假设通信代价不存在，即 $w(e) \equiv 0, \forall e \in E$. 为方便表示，我们将 $w(< t_i, t_j >)$ 简记为 $w_{i,j}$.

问题情境环境会有同质 (homogeneous) 环境与异质 (heterogenous) 环境的区别. 在同质环境中，任意一个任务，在任意一个处理器上运行所需的时间是相同的. 而异质环境中不同的处理器可能处理时间会不同. 在同质环境下，有 $v : \mathcal{T} \to \mathbb{R}^*$，对于 $\mathcal{T}$ 中的每一个顶点 t_i，$v(t_i)$ 表示该任务执行所需要的时间，不妨将其简记为 v_i，$i = 1, 2, ..., n$；在异质环境下，有 $v : \mathcal{T} \to (\mathbb{R}^*)^p$，对于 $\mathcal{T}$ 中的每一个顶点 t_i，$v(t_i)$ 是一个 p 维实向量，$v(t_i)$ 的第 j 个分量表示任务 t_i 在处理器 j 上的处理时间，不妨将其简记为 $v_{i,j}$，$i = 1, 2, ..., n,\ j = 1, 2, ..., p$.

$\mathcal{D}$ 中的一条路径 φ 指的是 $\mathcal{T}$ 中的顶点组成的一个序列 $t_{i_1}, t_{i_2}, \ldots, t_{i_s}$，满足

$$< t_{i_k}, t_{i_{k+1}} > \in E,\ k = 1, 2, \cdots, s-1$$

即任务 $t_{i_{k+1}}$ 对任务 t_{i_k} 有依赖关系. 其中 $s \in \mathbb{N}^*$ 表示路径 φ 中的顶点个数，称为路径的长度，记作 $L(\varphi)$，处理完成该路径上所有任务的总时间称为路径 φ 的时间，记作 $T(\varphi)$. 任务 t_{i_1} 称为路径 φ 的首任务，任务 t_{i_s} 称为路径 φ 的末任务. 对于任意两个任务 t_i, t_j，如果存在一条以 t_i 为首任务且以 t_j 为末任务的路径，则称 t_i 是 t_j 的祖先，t_j 是 t_i 的子孙.

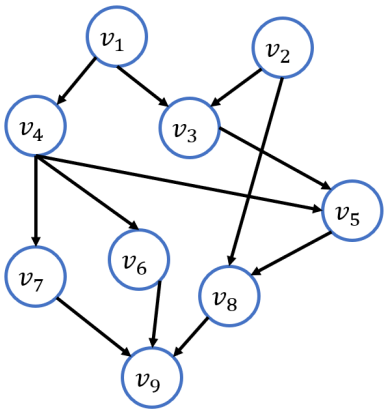


图 1　有向无环图 (DAG)

对于 $\mathcal{T}$ 中一个给定的任务 t_i，它的前向路径定义为以它为尾任务的所有路径中顶点个数最多的那一条，这条路径的长度称为它的前向路径长度，记作 ls_i；后向路径定

义为以它为首任务的所有路径中顶点个数最多的那一条，这条路径的长度称为它的*后向路径长度*，记作 lt_i. 显然后向路径不一定是唯一的. 另外，我们用 $p(j)$ 来表示后向路径长度为 j 的任务的个数，即

$$p(j) = \left|\{t_k \in V \mid lt_k = j\}\right|,\ j = 1, 2, \cdots, \mathcal{L}.$$

特别地，我们将图 $\mathcal{D}$ 中所有路径中长度最大的那一条称为*最长路径*，记它的长度为 $\mathcal{L}$，显然最长路径也不一定是唯一的，并且有 $lt_i \leqslant \mathcal{L},\ i = 1, 2, \ldots, n$. 例如，在**图 1** 中，如果在一个同质环境下，所有任务的处理时间都为 1，那么 t_4, t_5, t_8, t_9 是 t_4 的一条后向路径，故 $l_4 = 4$. 计算图中所有结点的后向路径长度后可知 $p(1) = 1,\ p(2) = 3,\ p(3) = 1,\ p(4) = 2,\ p(5) = 2$. t_1, t_3, t_5, t_8, t_9 是一条最长路径，从而 $\mathcal{L} = 5$.

一个*调度*是一个 m 维向量 $S =< s_1, s_2, ..., s_m >$，其中 $s_i = \{t_{i_1}, t_{i_2}, ..., t_{i_{n_i}}\}$ 是一个有序集合，表示处理器 p_i 上按时间顺序依次处理了 t_{i_1},t_{i_2},...,$t_{i_{n_i}}$ 这共计 n_i 个任务. 对于某一任务 $t_{j_l} \in s_j$，l 代表了在当前调度 S 下该任务在处理器 p_j 上的*优先级别*. 据此，我们可以定义两个函数 $p(t_i, S)$ 和 $r(t_i, S)$，分别表示在调度 S 下任务 t_i 所被分配的处理器及其在处理器上的优先级别.

调度有合法与非法之分. 我们称一个调度 S 是*合法的*，当且仅当经过有限长的时间后，按照该调度规定的优先级别可以执行完成所有任务. 易知，如果 $\exists s_k \in S, s_k = \{..., t_{k_i}, ..., t_{k_j}, ...\}$，且 $< t_{k_j}, t_{k_i} > \in E$，则在该调度 S 下必不可能完成所有任务，因为任务 t_{k_i} 永远无法开始进行，因此该调度是非法的. 以下讨论中，若无特殊说明，我们考虑的都是合法的调度.

处理完所有任务所需要的时间将会是一个确定值，称之为当前调度 S 下*完工时间* (makespan), 记作 $T(S)$. 具体来说，调度问题的优化原则较为多样化，除了完工时间外，还有能量消耗，负载平衡，系统可靠性，通信代价，处理器数目，系统温度，并行性，资金代价等. 但完工时间是最主要、也是研究最悠久最广泛的优化目标.

1 . 3 问题的难度

如果以完工时间作为优化对象，则该问题是一个典型的组合优化问题. 该问题所对应的判定问题可以表述为：输入一个任务图 $\mathcal{D} = (\mathcal{T}, E, v, w)$，输入处理器个数 $p > 1, p \in \mathbb{N}^*$，给定一个界 $K \in \mathbb{R}^+$，是否存在一个合法的调度 σ，使得 $T(\sigma) < K$？我们将这个判定问题记作 Dec. 如果处理器个数 p 不是输入的参数，而是一个提前已知的整数，则其对应的问题记作 $Dec(p)$. 问题的难度不加证明地总结如下 [1]：

1. 当 $E = \varnothing$ 时，即没有约束关系时，$Dec(2)$ 是 NP$-$ 完全的，不过存在伪多项式时间的算法. 进一步而言，$\forall \varepsilon > 0$，此时 $Dec(2)$ 还存在近似比为 $(1 + \varepsilon)$ 的以 $\frac{1}{\varepsilon}$ 的多项式为时间复杂度的近似算法，即完全多项式时间近似方案.

2. 当 $E = \varnothing$ 时，Dec 是强 NP− 完全的，即使输入中的所有参数在输入长度上都由某个多项式限定，这个问题仍然是 NP− 完全的，不存在伪多项式时间的算法.

3. 当 $E \neq \varnothing$ 时，$Dec(2)$ 是强 NP− 完全的，因而 Dec 是强 NP− 完全的.

从上述总结中可以看出，在不事先给定处理器个数 p (而是将其作为输入参数) 的条件下，无论有没有约束关系，该问题都是一个强 NP− 完全的问题，因此问题的难度可见一斑. 面对强 NP− 完全的问题，启发式的算法往往是值得考虑和研究的算法.

二、 传统启发式算法的应用和分析

2 . 1 基本思想

在设计调度策略时，我们很自然地想到去设计一个贪心的策略. 这里的贪心指的是，程序不会在某一任务可以开始执行时仍然等待一段时间，而是会尽可能早的开始执行每个任务. 在贪心的前提下，给定一个前文所定义的一个调度后，调度的完工时间就是唯一确定的. 给定任意一个调度后，假设我们不按照贪心的策略去执行，而是在某些时间刻意浪费处理器资源，那么当我们采取贪心策略、不浪费这些处理器资源后，总的完工时间一定不会变长. 这就表明，贪心的策略一定能够得到最优解. 不过，在贪心的前提下，如何生成一个好的调度，依然是比较棘手的问题. 直觉上，只要一个调度是在贪心的条件下执行的，那么它的完工时间一定不会太长. 下面，我们就来证明这个直觉.

引理 1　记 T_{max} 为最长路径花费的时间，则对于任意一个合法的调度 σ，有

$$T(\sigma) \geqslant T_{max}.$$

证明　设 Φ 是 G 中的一条最长路径，则 σ 必须依次在不同的时间执行 Φ 上的所有任务，从而 $T(\sigma) \geqslant T(\Phi) = T_{max}$. □

引理 2　设 Time 表示所有任务的处理时间之和，则对于任意一个合法的调度 σ，有

$$T(\sigma) \geqslant \frac{Time}{m}.$$

证明　由于 $T(\sigma)$ 表示调度安排中最后一个处理区完成任务的时间长度，m 个处理器在 $T(\sigma)$ 的时间内最多运行 $mT(\sigma)$ 时间，而由于 σ 最终完成了所有任务，故 $mT(\sigma) \geqslant Time$，即

$$T(\sigma) \geqslant \frac{Time}{m}.$$

□

定理 1 [2]　设某一任务调度问题由一有向无环图 $G = (V, E)$ 表示，假设求解该问题可调用的处理器总数为 m. 设 σ 是一个合法的调度，如果 σ 是贪心的，那么有

$$T(\sigma) \leqslant (2 - \frac{1}{m})T_{opt}.$$

证明 首先证明，存在 G 中的一条路径 φ，使得 $Free \leqslant (m-1)T(\varphi)$，其中 $Free$ 代表在完成整个调度的过程中 m 个处理器累计空闲的时间.

不妨将图 G 中某一任务的前驱任务、前驱任务的前驱任务等统称为该任务的祖先任务. 设任务 α_0 表示在当前调度下最后一个处理完成的任务，那么显然有 $\sigma(\alpha_0)+v(\alpha_0)=T(\sigma)$. 设 t_0 表示一个小于 $\sigma(\alpha_0)$ 的最大时间，使得在时间区间 $[t_0, t_0+v(\alpha_0)]$ 内存在空闲的处理器. 若这样的 t_0 不存在，则令 $t_0=-1$.

由于存在空闲的处理器，因此在 t_0 时刻必然存在 α_0 的某个祖先 α_1 正在被处理，否则 α_0 可以在更早的时刻开始处理，与调度是贪心的矛盾. 另外，从 t_0 的定义可知，从时刻 t_0+1 开始至时刻 $\sigma(\alpha_0)$ 为止，所有的处理器都处于工作状态. 重复上述过程，直至 $t_0=-1$，我们可以得到一个任务序列 $\alpha_k, \alpha_{k-1}, ..., \alpha_1$. 序列上的所有任务都在某条路径 φ 上，且存在处理器空闲的情形只可能出现在此序列上的某一元素正在被处理的时候，而当 $t=\sigma(\alpha_i)$ 时，由于 α_i 正在被处理，故至少有一台处理器正在工作，于是可以推出：

$$Free \leqslant (m-1)\cdot T(\varphi) \leqslant (m-1)T_{max}.$$

又由于所有处理器空闲时间和工作时间的总和应为 $mT(\sigma)$，故 $Free+Time=mT(\sigma)$. 又由引理 1，对任意合法的调度 π，有 $T(\pi) \geqslant T_{max}$, 从而 $T_{opt} \geqslant T_{max} \geqslant T(\varphi)$. 再由引理 2，对任意合法的调度 π，有 $Time \leqslant mT(\pi)$，从而 $Time \leqslant mT_{opt}$. 故有 $mT(\sigma)=Free+Time \leqslant (m-1)T(\varphi)+mT_{opt} \leqslant (2m-1)T_{opt}$，因而有

$$T(\sigma) \leqslant (2-\frac{1}{m})T_{opt}.$$

□

定理 1 证明了我们的猜想. 它表明：只要我们不有意浪费处理资源（即保证调度的贪心），我们得到的调度方案一定不会太差，即这一调度方案所需时间不会超过最优方案所需时间 T_{opt} 的两倍. 这是一个强有力的结论，许多后续的算法证明都是在这个结论的基础上继续优化得到的.

2.2 传统启发式算法

鉴于该问题是NP-完全问题,研究者广泛应用启发式算法,以期望获得一个相对优秀的解. 传统的启发式算法大致可以分为三类：基于列表 (List-based)、基于聚类 (Clustering-based) 和基于副本 (Duplication-based). 其中代表的经典算法可以用图 2 总结：

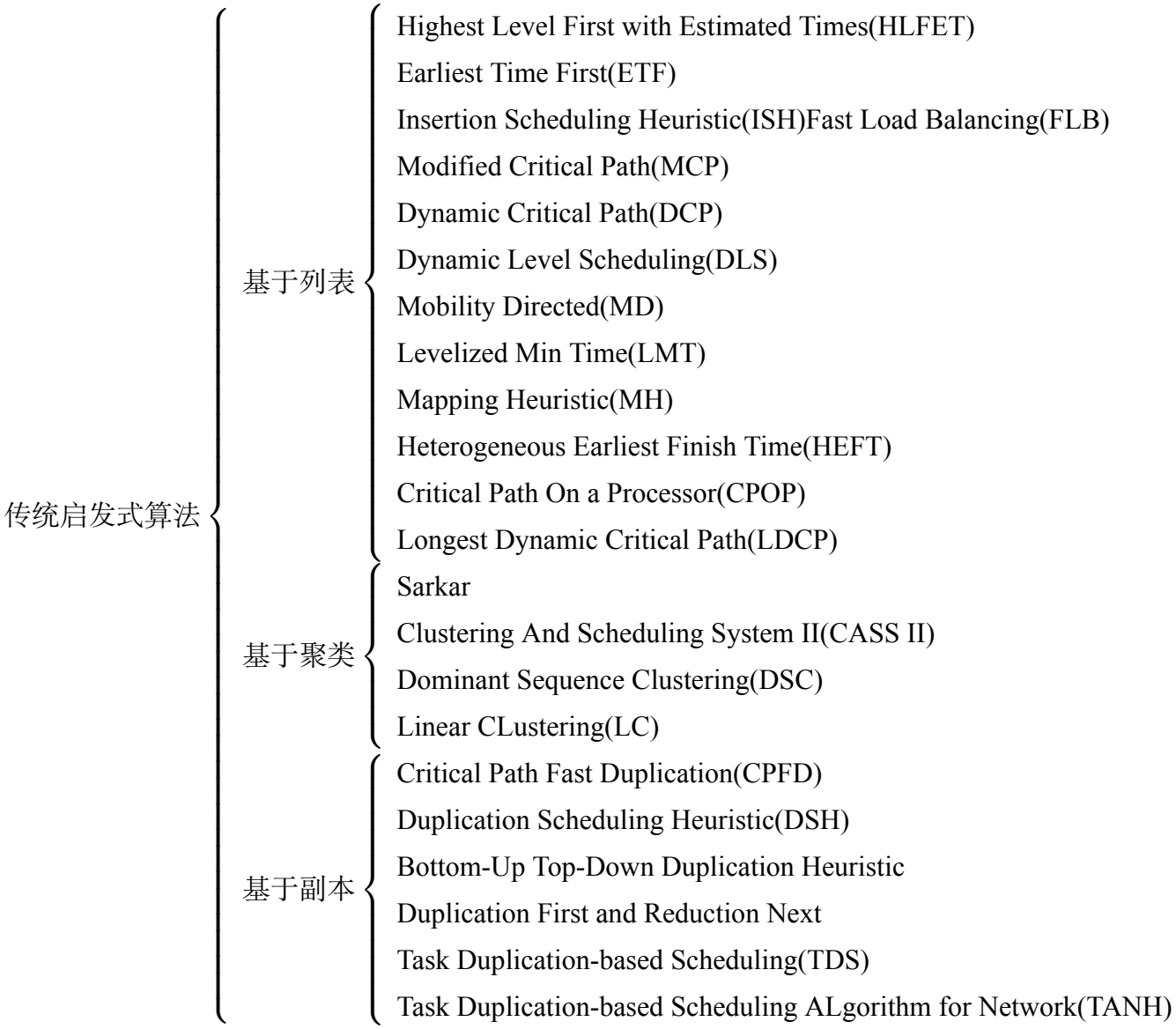


图 2　传统启发式算法的分类与经典算法

基于列表 (List-based) 算法中, 每个任务会被分配一个优先级, 算法一般会按照优先级的偏好将任务组成一个列表, 按顺序选择任务. 为某个任务选择处理器同理, 会按照某种偏好性选择优先级最高的处理器.1999 年, Topcuoglu Haluk 等人在论文 *Task Scheduling Algorithms for Heterogeneous Processors* [3] 中提出了两种代表性算法: Heterogeneous-Earliest-Finish-Time(HEFT) 和 Critical-Path-On-a-Processor(CPOP). 下面我们简析这两种算法的思想.

不妨用 $EST(t_i, p_j)$ 和 $EFT(t_i, p_j)$ 表示在处理器 p_j 上任务 t_i 的执行最早开始时间和最早结束时间. 定义如下:

$$EST(t_i, p_j) = \max\{T\_Available[j], \max_{t_m \in pred(t_i)}(EFT(t_m, p_k) + w_{m,j})\}$$

$$EFT(t_i, p_j) = v_{i,j} + EST(t_i, p_j)$$

其中，$pred(t_i)$ 表示的是任务 t_i 的所有直接前驱组成的集合，$T\_Available[j]$ 是处理器 p_j 可以执行任务的最早时间.

关键路径 (critical path,CP) 指的是 DAG 中最起点到终点最长的路径. 我们再定义两种优先级：

$$rank_u(t_i) = \overline{v_i} + \max_{t_j \in succ(t_i)} (\overline{w_{i,j}} + rank_u(t_j))$$

$$rank_d(t_i) = \max_{t_j \in pred(t_i)} (\overline{v_j} + \overline{w_{j,i}} + rank_d(t_j))$$

其中 $succ(t_i)$ 指的是任务 t_i 的直接后继组成的集合.

HEFT 算法　在任务排序阶段，每个任务会依据自身的运行时间和后继任务的通信时间，计算出 $rank_u(t_i)$, 并按照优先级降序排列生成任务的一个列表. 如果有两个任务的优先级相等，则随意安排一种顺序排列. 在任务分配阶段，算法整体采用最早完成时间的顺序将任务分配给处理器. 但 HEFT 的特殊之处在于，它额外采用了区间插入的技术. 即，对于每一个任务，考虑是否能插入一个更早的时间空隙中. 更早的时间空隙由两个已经在同一处理器上安排好的任务形成. 公式化地来讲，用 $list_{j,k+1}$ 表示已经分配给处理器 p_j 的第 k 个任务节点，则如果 t_i 可以被插入到处理器 p_j 中，p_j 需要满足下列关系式并具备最小的 k 值:

$$EST(list_{j,k+1}, p_j) - EFT(list_{j,k}, p_j) \geq v_{i,j}$$

HEFT 算法采用的区间插入技术，有效的利用了多核处理器的并行性，提高了利用率，进而提升了调度的整体效率. 但算法依然有一些明显的缺陷：任务排序时只考虑到与后继任务的通信开销，让一些后序值大的节点优先运行，但没有从图的整体拓扑结构比较任务的关键程度；利用区间插入技术插入到空闲时间段的任务数可能十分有限；任务间的通信代价可能很大，甚至远远超过了任务本身的运行时间等等. 整体来看，HEFT 算法的时间复杂度是 $O(T^2P)$.

CPOP 算法　针对 HEFT 算法的问题，原作者还提出了一种新的算法.CPOP 算法在任务排序阶段综合考虑了后序 $rank_u(t_i)$ 和前序 $rank_d(t_i)$, 加上任务自身的运行时间，计算每个任务节点的优先级，形成任务列表，并且得到一条关键路径. 然后在处理器中选择一个顺序执行关键路径上任务花费时间最短的处理器 CPP. 任务分配阶段，算法把关键路径上的任务分配给 CPP，其他任务结合区间插入技术，按处理器最早完成时间进行分配.

关键路径是图中最长的路径，决定了整个任务图的完成时间，而 CPOP 算法的目的就是给关键路径上的任务更高的优先级，保证关键任务能够得到优先调度且完成时间最快的处理器，来缩短所有任务的完成时间. 但 CPOP 算法同样有一些不足：只考虑了关键路径节点的优先级，忽视了不在关键路径上的关键节点的父节点，这类节点的优

先级很低，但它们的延迟又会影响关键路径的完成时间，导致算法调度效率的降低；同时，这种分配方式导致执行关键路径任务的处理器 CPP 极易负载偏重，而其他任务处理器利用率又不高，从而降低算法的并行性，导致处理器负载不均情况的出现. 整体来看，CPOP 算法的复杂度也是 $O(T^2P)$ 的.

定义 Schedule Length Ratio(SLR) 表示完工时间与理想最佳情况的比值，Speedup 表示简单程序的输出与完工时间的比：

$$SLR = \frac{makespan}{\sum_{n_i \in CP_{MIN}} \min_{j \in \mathcal{P}}\{w_{i,j}\}}\ ,\ Speedup = \frac{\min_{j \in \mathcal{P}}(\sum_{n_i \in DAG} w_{i,j})}{makespan}$$

根据论文中的实验结果，HEFT 和 CPOP 在 SLR、Speedup、运行时间三个指标上都有很好的表现，其中 HEFT 更是在三项指标上全面优于其他算法，在 20 个节点的时候 SLR 达到了 3 左右，Speedup 达到了 1.8 左右；其他算法例如 LMT 对应的值分别是 5 和 1.4.

基于聚类 (Clustering-based) 算法适用于并行分布式的系统，能够有效减少图的通信延迟. 在这种算法中，彼此有着较强联系和较高通信代价的任务会被倾向于放置在一个聚类中，并被分配给一个单独的处理器.Yang Tao 在论文 *DSC: Scheduling Parallel Tasks on an Unbounded Number of Processors* [4] 中提出了 Dominant Sequence Clustering(DSC) 算法.

DSC 算法　记 $tlevel(t_i)$ 表示从一个起点到 t_i 的最长路径的长度，$blevel(t_i)$ 表示从 t_i 到一个终点的最长路径长度. 起点和终点分别表示入度为 0 和出度为 0 的点. 每个点的优先级定义如下：

$$PRIO(t_i) = tlevel(t_i) + blevel(t_i)$$

算法将所有任务初始化为各自独立的 UEG(unexamined part) 单元 m 每次选择 PRIO 最大的 UEG 任务进行检查，将其标记为 EG(examined part), 通过把边权修改为 0 将其与 EG 的任务合并，然后更新 UEG 节点的 PRIO 值. 这样的选择顺序与拓扑的遍历图是等价的. 而按照拓扑序列选择任务的原因是, 每次合并操作将边置 0 后, 只需要计算当前正在检查的任务的所有直接后继，节省了运算的次数. 算法整体复杂度是 $O((T+E)logT)$. 实验效果方面，在 180 个随机生成的 DAG 上 DSC 的性能比 ETF 大约提升了 1.91%, 与 Sarkar 相比则平均提升了 15% 左右.

基于副本 (Duplication-based) 的算法通过在不同的处理器上使用任务副本，可以减少调度整体的运行时间. 在这种方法中，不只在一个处理器上执行某个任务，避免了不同处理器之间相关任务的运行信息的传播发送，从而降低了通信开销.Ishfaq Ahmad 在论文 *On Exploiting Task Duplication in Parallel Program Scheduling* [5] 中提出了基于副本的经典算法：Critical Path Fast Duplication(CPFD).

CPFD 算法　首先，我们将 DAG 中的顶点分为三类：CPN，在关键路径上的点；IBN，不在关键路径上但存在路线到达 CPN 的点；OBN，既不是 CPN 又不是 OBN 的点. 然后按如下方式构造 CPN-主列表：对于一个 CPN，如果它所有的父节点都已经在

列表中，则直接将其加入主列表中. 否则递归的寻找 $blevel$ 值最大的点 t，若 t 的所有父节点都在列表中则加入列表，否则继续寻找父节点.$blevel$ 值相等时选择 $tlevel$ 较小的那个. 完成递归后将 CPN 加入列表中. 最后，将所有的 OBN 按照 $blevel$ 递减的顺序加入 CPN-主列表中. 这样保证了 CPN 不会违反任务的约束关系，而 OBN 按照拓扑序加入也保证了在列表中前驱 OBN 总在后继 OBN 之前.

任务副本技术方面，算法采用了类似于区间插入技术的方式. 一个节点的开始时间是由父节点数据到达的时间决定的. 我们把将父节点中数据到达时间最迟的那个称作是该节点的 Very Important Parent(VIP). 期望是能尽早的开始一个节点的运行，那么就要提前安置该节点的 VIP，这就要求在某个处理器上找到一个合适的时间槽. 算法会扫描整个处理器运行的时间轴，来找到足够安置 VIP 的不违反约束关系的最早的时间槽. 具体的判断方式与 HEFT 算法中插入任务的方式相似.

CPFD 算法的整体过程如下：

1) 选定一条关键路径 CP，构建 CPN-主列表.

2) 选定候选节点为主列表中第一个没有调度的 CPN.

3) 记包括所有保有候选节点父节点的处理器和一个空处理器的集合为 P_SET.

4) 对 P_SET 中的每个处理器，利用副本技术求出候选节点的最早开始时间.

5) 选择最早开始时间最小的处理器 p，将候选节点调度给 p.

6) 重复 2)-5)，直到主列表中的所有 CPN 已完成调度.

7) 对主列表中的每个 OBN，重复 2)-5).

经分析可得，CPFD 算法的时间复杂度为 $O(T^2E)$. 实验效果方面，50 个节点时，CPFD 在 NSL(调度长度与 CP 长度之比) 上达到了 1.2 左右，500 个节点时依然保持在 1.6 以下. 在作者测试的 600 个 DAG 中，CPFD 算法的输出与最佳时间相同的达到了 551 个，比例高达 91.8%，体现了算法优秀的表现.

三、 遗传算法的复现和实验

3 . 1 遗传算法简介

遗传算法 (Genetic Algorithm,GA) 是元启发式 (Meta-Heuristic) 算法的一种. 第二部分中所述的传统启发式策略在寻找特定问题特定实例的最优解的过程中依据当前全局或局部的经验来改变其搜索路径，而元启发式策略则通常是一个更为通用的启发式策略，其不借助于某种问题的特有条件，从而能够运用于更广泛的方面，也能够很好地避免搜索陷入局部最优. 因此，**图 2** 中展示的传统启发式算法是针对“多处理器上的任务调度问题”所特制的，而以遗传算法为代表的元启发式算法是一种独立于问题的算法，将遗传算法应用于某一个具体问题时则需要对于其中的组件和操作做出相应的规定和限制.

遗传算法是模拟达尔文生物进化论的自然选择和孟德尔遗传学机理的生物进化过程的计算模型，是一种通过模拟自然进化过程搜索最优解的方法. 其从一组可行解 (称之为*种群*) 和一组基于种群本身定义的“生物算子”开始. 在每次的迭代中，通过*繁殖*操作创造出一个新的解决方案集合，即新的种群. 根据进化理论，只有种群中“最合适生存”的元素才最有可能存活并产生后代，从而将其所带有的“遗传物质”传递给下一代. 因此，遗传算法的运行经历了一个简单的循环：创建种群，评估种群中每个体，选择优良个体进行繁殖从而创建新种群.

种群中的每个个体被唯一地编码为某一字母表中元素组成的字符串，可称之为*基因型*. 每一个基因型会被一个*目标函数*唯一地映射到一个*表现型*. 表现型的值往往是特定于问题的意义，比如表示个体的适应程度. 适应程度高的优良个体将被保留，繁殖后代.

繁殖涉及两类基本的遗传操作，即*交叉* (crossover) 和*变异* (mutation). 最简单的交叉算子是单点交叉，即两个个体的基因型在随机位置后交换，产生两个新的后代. 突变算子则以一定的概率作用于所有新产生的个体的基因型，其可被视为一种背景算子，确保找到最优解的概率总不为零. 突变往往会抑制收敛到具备最优解而不是全局最优解的可能性.

繁殖完成后，重复上述循环. 对新产生的个体，使用目标函数给出表现型 (适应度值)，根据适应度选择个体进行繁殖. 随着优秀个体的保存和繁殖、拙劣个体的消亡，种群中个体的平均表现预计会提高. 遗传算法在给定的条件下终止，例如已完成一定数量的世代、已达到一定程度的适应度或已收敛到搜索空间中的一个点.

综上所述，对于任何问题的遗传算法的构建都可以分为如下五个不同但又相关的要素：

1. 基因型字符串表示形式的选择.
2. 初始种群的创建.
3. 交叉、变异等遗传算子的构造.
4. 目标函数 (适应度函数) 的确定.
5. 遗传算子控制概率的确定.

以上五部分都对遗传算法的求解结果和性能有很大的影响. 因此在设计“多处理器上的任务调度问题”的遗传算法时，此五项也是我们重点关注、并且可以进行多样化的修改的.

3.2 本问题的一种遗传调度算法

通过梳理“多处理器上的任务调度问题”前人的研究脉络，我们关注到了 1994 年 2 月 E.S.H. Hou, N. Ansari 和 Hong Ren 在期刊 IEEE Transactions on Parallel and Distributed Systems 上发表的论文 *A Genetic Algorithm for Multiprocessor Scheduling* [6]. 根据我们的调查，这篇论文是遗传算法在本问题中应用的“鼻祖”，自此之后传统的启发式算法就

少有论文发表，而一系列多种多样的使用遗传算法的论文在各类国际期刊杂志上涌现. 因此，我们针对这篇论文所提出的具有跨越性意义的算法进行了复现，通过理论分析和实验检验分析该算法的优点和缺点，进而提出了一些可供改进的方案，并在改进的基础上进行了实验对比.

这篇文章考虑的是同质环境下的任务调度问题，有无通信代价对于文章中的算法并没有实质性的影响. 下面，我们从上文所述的遗传算法的要素的角度阐述该算法的基本思想：

3.2.1 基因型编码

种群中的每个个体 S 都表示一种调度，因此每个个体的基因型包含 m 个字符串，即 $s_1, s_2, ..., s_m$. 处理器和字符串之间是一一对应的关系，其中每个字符串表示调度到某个特定处理器的任务. 这种表示形式与 1.2 中所述的调度的表示方式是一致的. 不过，很显然，这种使用字符串的编码方案也可能会表示一个不满足优先约束的调度，即非法的调度. 因此，需要找到一种方式来确保在种群中的每个个体始终都表示一个合法的调度，不论种群是初始种群，还是经历若干轮繁殖、变异之后的种群.

一种可行的策略是，为每个任务分配一个高度值，通过高度的限制来确保调度的合法性. 具体而言，对于一个任务 t_i，设其所有直接前驱结点的前向路径长度中的最大值为 $hp(t_i)$，其所有直接后继结点的前向路径长度中的最大值为 $hs(t_i)$，于是取 t_i 的高度值 $height(t_i)$ 为开区间 $(hp(t_i), hs(t_i))$ 中的一个随机整数. 可以看出，任务的高度值诱导了任务的部分顺序，这有助于用优先关系表示任务依赖关系. 例如，如果任务 t_{i_1} 是任务 t_{i_2} 的祖先，则必有 $height(t_{i_1}) < height(t_{i_2})$. 因此，如果每个处理器上的任务的执行顺序都是按照高度值从低到高的次序排列的，则由此得到的调度一定是合法的调度.

3.2.2 初始种群的创建

初始种群的生成将利用前文引入的高度值. 其具体构建方法为：将所有具有相同高度值 h 的任务归入集合 $SH(h)$ 中，对于每个高度 $h = 0, 1, 2, ...$，按次生成闭区间 $[\,0, |SH(h)|\,]$ 中的随机整数 r，而后从 $SH(h)$ 中随机选出 r 个将其安排给处理器 p_1,，并将这 r 个任务从 $SH(h)$ 中删除. 重复上述操作，为 $p_2, p_3, ..., p_{m-1}$ 安排任务. 最后，将集合中还剩下的元素都安排给 p_m 即可. 可以看到，在这样的操作下每个处理器上的任务都会按照高度值递增 (不减) 的顺序排列，因此必然得到的是一个合法的调度.

若初始种群大小为 $\mathcal{M}$, 则调用 $\mathcal{M}$ 次上述算法，生成 $\mathcal{M}$ 个随机的初始调度.

3.2.3 适应度和遗传算子

对于种群中的每个个体，其适应度反映了该个体的优良程度. 因此，我们将每个个体 (即每种调度安排)S 的适应度与该调度下的完工时间 $T(S)$ 相联系——完工时间越短，适应度越高. 对于一个调度 S, 其令适应度 $fitness(S) = C_{max} - T(S)$，其中 C_{max} 为目前已发现的最长的完工时间.

算法关心的遗传算子主要有三个. 一是选择算子，即怎样选择种群中的个体进行繁殖. 在实际算法复现中，为方便实现论文所述效果，我们采取的选择算子如下：为种群中的每个个体分配一个区间，假设个体为 $S_1, S_2, ..., S_{\mathcal{M}}$，则为 S_1 分配区间 $[1, fitness(S_1)]$，为 S_2 分配区间 $[fitness(S_1)+1, fitness(S_1)+fitness(S_2)]$，以此类推. 而后，随机生成一个闭区间 $[1, \sum_{i=1}^{\mathcal{M}} fitness(S_i)]$ 上的随机数. 若随机数落在 S_k 对应的区间内，则选择出个体 S_k. 显然，适应度更高的个体被选择的概率更大.

二是交叉算子，即怎样对选择出的两个个体 (父代) 进行杂交，产生新的两个个体 (子代). 对于两个父代调度而言，随机选择一个高度值 h，将父代调度中每个处理器上高度值大于 h 的任务对调，而高度值小于等于 h 的保持不变. 这样，就形成了两个依然合法的子代调度.

三是变异算子，即一个个体怎样进行变异. 在一个调度中，我们任意交换两个高度值相同的调度所在的位置，这样得到的新调度依然合法，故可将这个操作视为变异操作.

确定了上述各项后，**Algorithm 1** 给出了算法的整体框架.

Algorithm 1 Genetic Algorithm

Input: A DAG application; Crossover Probability; Mutation Probability; Size of Run; Size of Population; Number of Processors

Output: A Task Schedule;

1: Initial Population **with** *Size of Population*
2: **for** *Size of Run* iterations **do**
3: Select
4: Crossover **with** *Crossover Probability*
5: Mutation **with** *Mutation Probability*
6: **end for**
7: **return** best schedule in population

3.3 实验结果及分析

按照 3.2 中所述的算法设计思想，我们以 Python 为程序设计语言成功复现了该遗传调度算法，算法代码见附件中 Genetic_basic.py[1]. 为了验证该算法的实际效果，我们使用

[1]代码也在 https://github.com/yyChen233/Algorithm-Design-and-Analysis-Project 上开源

了数据集 Standard Task Graph Set[2]，该数据集中提供了多达 2700 个随机的任务图，图形大小 (任务数量) 在 50 至 5000 之间变化. 此外，数据集中还提供了三个从实际应用中建模的标准任务图，其中包括包含 88 个结点、131 条边的“机器人控制”任务图，而原论文中也使用了这个任务图来评估算法.

基于该数据库，我们对复现的遗传调度算法进行了实验. 数据集中对于每一个任务图都给出了通过暴力搜索算法得出的最优调度的完工时间或最优完工时间的上下界，因此通过对比算法给出的调度的完工时间与最优的完工时间，就能够得出算法在给出优良调度方面的性能. 设对于多处理器上的任务调度问题的任意实例 σ，遗传算法给出的调度所用的平均完工时间为 T_{GA}，最优调度的完工时间为 T_{opt}，定义偏离比为 $\frac{T_{GA}-T_{opt}}{T_{opt}} \times 100\%$，偏离比越小则意味着遗传算法给出的调度越好.

由于原文中进行测试的任务调度图的结点数也在 50 个左右，因此我们选择数据集上的 50 个结点的任务调度图进行测试. 我们随机挑选了 3 个图，每个图按照处理器数量为 2,4,8 分别进行 10 次重复测试，取测试结果中的最优结果. 此外，由于在原论文中包含“机器人控制”任务图的测试结果，我们也针对这一任务图按原文方式进行了测试.

作为对照，我们还编写了一个利用前文所述的关键路径为原理的基于列表 (List-Based) 的传统启发式算法 (代码见附件 List.py)，用传统启发式算法给出的结果 T_{List} 和遗传算法给出的结果 T_{GA} 进行对比.

在遗传算法中，初始种群的数量 $\mathcal{M}$、最多繁殖的代数 $\mathcal{G}$ 和突变率 $\mathcal{Q}$ 是三个需要人为设定的参数. 其中, 初始种群的数量和最多繁殖的代数对于最终结果的影响很大. 因此，在实验中, 我们固定 $\mathcal{Q} = 0.05$, 选取 $(\mathcal{M}, \mathcal{G}) = (20, 1000), (100, 100), (100, 1000), (200, 1000)$ 四种情况分别测试实验结果. 测试结果如下**表 1**:

根据实验结果，可以得出如下四点分析和猜测:

1. 遗传算法具有良好的效果. 遗传算法展现了其针对普遍问题的强大能力，我们仅仅通过简单地定义遗传算法中的种群、遗传算子等操作，通过初始种群的若干次迭代, 就可以得到能够媲美甚至超越针对该问题特制的传统启发式算法. 在遗传算法中并没有过多的使用诸如前后向路径长度等有向无环图中的信息，因此算法得出的效果超出了我们的预期.

2. 遗传算法的效果十分依赖于初始种群的数量 $\mathcal{M}$ 和最多繁殖的代数 $\mathcal{G}$ 两个参数. 一般来说，初始种群的大小越大，得到的结果越好，因为初始种群中更有可能出现表现型优异的个体；繁殖代数越多，得到的结果越好，因为繁殖会在“适者生存”的条件下让种群整体趋于更优. 通过实验数据可以观察到，初始种群相较于繁殖代数有着更为重要的地位. 容易发现，参数 $(\mathcal{M}, \mathcal{G}) = (20, 1000)$ 下得到的效果大概率劣于参数 $(\mathcal{M}, \mathcal{G}) = (100, 100)$ 下得到的效果，而后者相较于前者初始种群仅扩大了 5 倍，繁殖代数却减少了 10 倍. 在本问题中，通过实验结果可以明显观察

[2]数据集网址为 http://www.kasahara.cs.waseda.ac.jp/schedule/index.html

数据集	处理器数量	T_{opt}	T_{List}	T_{GA}			
				(20,1000)	(100,100)	(100,1000)	(200,1000)
rand0000	2	131	132	138	131	131	131
	4	66	72	101	85	83	77
	8	55	55	83	69	71	64
rand0096	2	249	250	253	251	249	249
	4	125	132	176	146	141	134
	8	63	79	134	117	111	111
rand0157	2	420	422	437	422	424	420
	4	411	411	430	415	411	411
	8	411	411	422	416	413	411
robot	2	1242	1267	1454	1331	1313	1296
	3	879	938	1176	1050	1048	1026
	4	659	796	980	947	923	897

表 1　实验结果 1

到，取参数 $(\mathcal{M},\mathcal{G}) = (20,1000)$ 的并不足以让遗传算法发挥出良好的性能，取参数 $(\mathcal{M},\mathcal{G}) = (100,1000)$ 或 $(200,1000)$ 能够以较大概率得到较好的结果，但是参数 $(\mathcal{M},\mathcal{G}) = (200,1000)$ 下算法运行时间较长，因此选取 $(\mathcal{M},\mathcal{G}) = (100,1000)$ 是相对最佳的参数选择. 不过事实上，论文中采取的参数设置为 $(\mathcal{M},\mathcal{G}) = (10,1000)$. 因此我们有理由怀疑论文选取的测试数据集十分特殊，不具有普遍性，因为在随机选取的测试集上采取 $(\mathcal{M},\mathcal{G}) = (10,1000)$ 的参数大概率无法得到较好的结果.

3. 遗传算法对于稠密图的效果比稀疏图要好. 测试集 robot 是一个有 88 个结点、131 条边的有向无环图，如果我们定义满足 $|E| < |V|\log|V|$ 条件的图 $G = (V,E)$ 为稀疏图，显然 robot 非常稀疏. 稀疏意味着任务与任务之间的约束条件少，故可行的调度方式就会很多. 因此在初始种群中，不一定会出现表现型非常优良的个体，从而在后面的繁殖过程中也就难以出现优良个体，因此难以得到较好的结果. 论文中针对 robot 数据集的测试结果要明显好于我们实际测试的结果，我们有理由怀疑在论文初始种群中人为加入了提前已知的表现型较好的个体，从而能够得到较好的结果.

4. 遗传算法对于处理器数量较少的实例的效果比处理器数量较多的实例更好. 从表中测试结果来看，处理器数量为 2 时，遗传算法基本上可以给出非常接近最优解的结果；处理器数量增加至 8 时，遗传算法给出的结果往往就会产生 15% 及以上的偏离比，这一点在稀疏图上表现得尤为明显. 事实上，在处理器数量很多时，高并行度可以导致某些特别的解下调度时间很短，而这些特别的解在遗传算法的初始种群中不一定出现，因此会导致偏离比的增加.

3.4 算法分析和改进策略

通过观察上述遗传算法的运行方式和实验结果，我们提出了该算法的几点缺陷.

首先，算法所生成的初始种群具有不平衡性. 事实上，对于该算法初始种群中的任意一个调度 S，在处理器 p_i 上处理的任务的数量的期望会大于在处理器 p_{i+1} 上处理的任务的数量的期望，$1 \leqslant i < m$. 任务分配算法的不均匀性到这了这一结果.

其次，算法存在搜索空间不完整性的缺陷. 交叉算子虽然简洁快速，但是存在一个严重的问题，即某些可行解 (合法的调度) 无法被生成. 因此，算法的搜索空间不能覆盖整个可行域，这就导致某些情况下无法生成比较优良的解.

例如，在如下**图 3** 所示的有向无环图中，设任务 $1 \sim 6$ 和任务 $8 \sim 10$ 的运行时间均为 1，任务 7 的运行时间为 10. 设有两个同质的处理器 p_1, p_2，忽略通信代价. 根据 3.2.1 中关于高度值的定义，有:

$$height(1) = height(2) = height(3) = 0$$

$$height(4) = height(5) = height(6) = 1$$

$$height(7) = height(8) = height(9) = 2$$

$$height(10) = 3$$

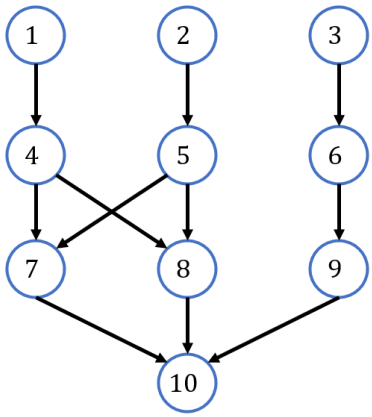


图 3　示例

示例的最优解为: p_1 上依次处理任务 $1,4,7,10$; p_2 上依次处理任务 $2,5,8,3,6,9$, 最优完工时间为 13. 然而, 在最优解中, p_2 上的任务并不满足高度值不减的顺序. 然而, 如果按照高度值不减的次序使得任务 3 在任务 4 和 5 之前进行, 那么任务 7 也将被推迟完成, 因此就无法达到完工时间为 13 的最优解了. 这就表明, 原论文中始终保持处理器上的任务按高度值不减的顺序排列可能会错过最优解. 再次, 算法具有一定的无知性. 算法中唯一的知识来源于 "高度值" 和 "适应度值", 这都是个体的结构性质. 换言之, 个体的 "素质" 没有得到较好的评估, 在繁殖后代的过程中也没有去有意识的保留父代较好的遗传性质.

事实上, 针对算法的 "无知性", 后续研究给出了大量的解决策略和方案. 其中很容易想到的解决方法就是将传统启发式算法的一部分思想移植到遗传算法中去. 传统启发式算法大多在算法运行过程中产生了对于有向无环图关键路径等属性的认知, 这些信息可以更好地帮助我们在遗传算法中繁殖出更好的后代. 后文中, 我们就将介绍一些增强的遗传调度算法的思想. 而在这一部分中, 我们着重关注于算法的前两个缺陷, 我们希望在不改变原论文算法基本框架的基础上对这两个缺陷的解决做一些研究.

针对第一个缺陷, 一种可行的解决方案是: 维护一个任务集合 F, 表示所有前驱任务业已完成的任务的集合. 从 F 中随机选择一个任务 t_i, 将其随机分配给一个处理器 p_j, 重复该过程直至所有任务都得到分配. 通过这种方法, 每个处理器上得到的任务的数量的期望值是相同的.

针对第二个缺陷, 考虑摒弃 "高度值" 的使用, 重新设计交叉算子. 事实上, 我们只需要将任务集合 $\mathcal{T}$ 划分成为两个不相交的集合 T_1, T_2, 保证有向无环图中没有从 T_2 中任务代表的结点指向 T_1 中任务代表的结点的有向边. 显然, 任何一个合法的调度中, T_1 中的任务必须在 T_2 中任务之前完成. 因此, 新的交叉算子就是保持父本 T_1 中的任务不变, 交换父本 T_2 中的任务.

设 $\mathcal{D} = (\mathcal{T}, E)$, 给定一个调度 S, 定义由 S 导出的有向图为 $\mathcal{D}(S) = (\mathcal{T}, E(S))$, 其中: $E(S) = E \cup \{(t_{i_1}, t_{i_2}) \mid (t_{i_1}, t_{i_2}) \notin E,\ p(t_{i_1}, S) = p(t_{i_2}, S) \ \wedge\ r(t_{i_1}, S) = r(t_{i_2}, S) - 1\}$. 再设 $E^+(S)$ 表示 $E(S)$ 的传递闭包, 则显然可以看出, 一个调度 S 是可行的当且仅当 $\mathcal{D}(S)$ 是无环的.

我们定义图 $(\mathcal{T}, E(S_1) \cup E(S_2))$ 表示由调度 S_1, S_2 导出的约束关系图, 于是可以得到如下的针对父本 S_1, S_2 划分 T_1, T_2 的方法如下:

1. 初始时令集合 $\mathcal{F} = \mathcal{T}$.
2. 随机选择 $\mathcal{F}$ 中的一个任务 t_i 以及一个集合 $T = T_j, j = 1$ 或 2.
3. 如果 $T = T_1$, 则有 $T_1 = T_1 \cup \{t_i\} \cup \{t_{i'} \in \mathcal{F} \mid (t_{i'}, t_i) \in (E(S_1) \cup E(S_2))^+\}$.
4. 否则, 有 $T_2 = T_2 \cup \{t_i\} \cup \{t_{i'} \in \mathcal{F} \mid (t_i, t_{i'}) \in (E(S_1) \cup E(S_2))^+\}$.
5. 将所有插入 T_1 或 T_2 中的结点从 $\mathcal{F}$ 中删去, 而后重复 $2 \sim 5$ 直至 $\mathcal{F}$ 为空.

在上述操作中, 如果 t_i 被插入 T_1, 则其在 $\mathcal{F}$ 中的所有祖先都将被插入 T_1 中; 如果

t_i 被插入 T_2，则其在 $\mathcal{F}$ 中的所有后继都将被插入 T_2 中. 这就保证了当 $\mathcal{F}=\varnothing$ 时，有向无环图中没有从 T_2 中任务代表的结点指向 T_1 中任务代表的结点的有向边.

我们依然利用 Python 为程序设计语言实现了上述改进算法，算法代码见附件中 Genetic_advance.py. 我们使用部分原先的数据集对改进算法进行了测试. 实验结果如下**表 2** 所示，其中 $T_{GA_{basic}}$ 表示改进前算法得出的结果，$T_{GA_{advance}}$ 表示改进后算法得出的结果. 实验结果表明：在设置参数 $(\mathcal{M},\mathcal{G})=(20,500)$ 下得到的结果与原先算法在 $(\mathcal{M},\mathcal{G})=(200,1000)$ 下得到结果相当，这表明改进的算法是有一定效果的. 不过，由于初始种群依然是随机化的，因此改进算法也无法保证找到最优解，在稀疏图中的表现和原算法一样依然不是非常良好. 此外，改进算法中，由于划分 T_1,T_2 的过程较为繁琐，改进算法的执行时间远高于原算法，大约为原算法的 $10\sim100$ 倍. 因此，改进的算法不具有太强的实用性，不过它确实实现了搜索空间的全覆盖，这一点是有意义的.

数据集	处理器数量	T_{opt}	$T_{GA_{advance}}$	$T_{GA_{basic}}$	
			(20,500)	(20,1000)	(200,1000)
rand0067	4	293	307	337	295
	8	293	318	333	308
rand0096	4	125	138	176	134
	8	63	117	134	111

表 2　实验结果 2

总而言之，单纯使用包含随机因素的遗传算法可能无法达到媲美传统启发式算法的效果，因为遗传算法的随机初始化十分影响算法效果. 不过，即便如此，遗传算法在一些稠密图上的表现也是相当惊人的，这充分说明了其作为通用的元启发式算法的强大能力. 如果将遗传算法中融入传统启发式算法的部分思想，所得到的调度算法就很有可能非常优异.

四、 一种增强的任务调度遗传算法

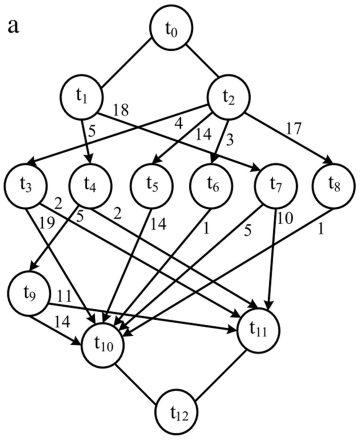
在 2017 年的 *An enhanced genetic algorithm with new operators for task scheduling in heterogeneous computing systems* [7] 一文中，作者提出了基于元启发式算法和遗传算法的算法. 该算法提出让初始种群尽可能地接近最优解，使得算法能够减少生成相对优化解所需的迭代次数. 该算法提出了一种新的反转算子. 这个操作符促进了每一代将产生的样品之间的多样性.

通过使用这个新的操作符，可以使生成的总体中重复调度的数量减少. 算法利用优先级对任务进行处理，用前文 2.2 中定义过的两个指标，$rank_u$ 和 $rank_d$ 衡量，其中每个节点的 $rank_d$ 表示该节点可能的最早开始时间，$rank_u$ 表示该节点到出口的可能最长传递时间，这样的两个参数成功的改进了以往的遗传算法单纯使用层次划分导致的局限性，对可行解的空间判断更加精准也对遗传算法生成的基因串的交叉和变异进行了扩张.

4.1 算法设计

本方案中同样包含三个基本的遗传操作：选择、交叉和突变. 与前文所述的遗传算法不同的是：在 EGA-TS 算法中，初始种群的生成并不是完全随机的而是经过均衡化处理的更优的时间表，遗传过程中的操作符也经过了调整，并加入了更有利的操作符.

a



b

task	p0	p1
t0	0	0
t1	4	6
t2	15	22.5
t3	4	6
t4	13	19.5
t5	10	15
t6	7	10.5
t7	8	12
t8	4	6
t9	12	18
t10	6	9
t11	9	13.5
t12	0	0

图 4　(a) 一个任务依赖关系及通讯代价实例
(b) 任务在不同处理器上的消耗

为方便描述算法思想，补充定义一些数学符号. 记没有前驱的任务为 T_{entry}，没有后继的任务为 T_{exit}. 由于我们可以在图中增加虚拟的 “超级源点” 和 “超级汇点”，因此总可以保证 T_{entry} 和 T_{exit} 的唯一性. 设 NL 表示节点优先级，即节点的 $rank_u$ 和 $rank_d$ 的总和. 记 CP 表示关键路径，LBP 表示当前所选的处理器，PC 为处理器计数器. 其余定义同 1.2 中的定义.

4.1.1 初始化种群

传统的遗传算法初始化种群的方案选用随机处理，没有尝试让初始种群更优，可能会生成一组并不适用于当前搜索空间的可行解. 在本算法中，启发式算协助遗传算法获取更为优秀的初始种群，从而更快速准确的在合理的时长内获得优化解.

算法预先通过分割将调度分为多个部分，每个部分包含的是能够在不同的处理器上并行执行的独立任务（两者之间没有任何的前驱或后继关系）. 分割后启发式的选择一

组更接近关键路径的任务，分配给最快的处理器执行，其余的任务交给其所属任务组对应的处理器，这样就能够生成一组更加接近最优情况的初始种群. 从而在更大限度上保证初始种群的完备性.

Seg 1	t1	t2				
Seg 2	t4	t7	t6	t3	t5	t8
Seg 3	t9					
Seg 4	t11	t10				
Seg 5	t12					

图 5　分割算法的分割结果

在该初始化算法中，采取算法的伪码参考 **Algorithm 2**.

Algorithm 2 InitialPopulation

Input: A DAG application

Output: The initial population

```
Segmentation
repeat
    for segment generated by Segmentation do
        for nodes in a segment do
            if NL = |CP| then
                Assign it to the fastest processor
            else
                Assign it to the LBP
            end if
        end for
    end for
until Size of Population
```

其中的 NL 和 CP 可以通过 (1) 式、(2) 式在多项式时间内求解

$$NL = rank_u(t_i) + rank_d(t_i) \tag{1}$$

$$|CP| = [rank_u(T_{entry}) + rank_d(T_{entry})] \; or \; [rank_u(T_{exit}) + rank_d(T_{exit})] \tag{2}$$

而 $rank_u$ 和 $rank_d$ 的数值可由 2.2 中给出的方程在多项式时间内获取：

$$rank_u(t_i) = \overline{v_i} + \max_{t_j \in succ(t_i)} (\overline{w_{i,j}} + rank_u(t_j))$$

$$rank_d(t_i) = \max_{t_j \in pred(t_i)} (\overline{v_j} + \overline{w_{j,i}} + rank_d(t_j))$$

其中对 LBP 的限定由 (3) 式给出.

$$LBP = PC \mod m \tag{3}$$

4.1.2 种群筛选

现今遗传算法的选择已经具有多种模式，在此类问题中已经得到尝试的有轮盘赌选择、蒙特卡洛选择、排序选择和精英选择. 在 2015 年的实验中，发现轮盘赌选择模式对 DAG 上的任务调度问题似乎能起到更好的作用. 在这里，需要利用 fitness 辅助，利用完工时间 (makespan) 对方案进行衡量：

$$makespan = AFT(T_{exit}), \tag{4}$$

其中的 AFT_{exit} 表示该调度方案最后一个结束的时间. 借助这个参数, 可以得到该基因串的适应度 fitness，这里简单的取为 $(\max\{makespan\} - makespan_i + 1)$ 即可. 而算法这一步对轮盘赌的选择模式可能会是种群筛选成功程度的决定性因素. 将该方案与他之前的所有方案中的适应度值比较实行轮盘赌选择模式，从而尽可能地选择种群中最优有能保证其存在不确定基因串的种群进行遗传. 这种局部轮盘赌的模式既保证了算法执行过程中种群中最好的一部分会被保留，剩余的不够好的部分中也有一定的更为合理的概率被留存，降低了因局部的不优导致其较优的部分未能被遗传下去的可能性. 算法的具体过程参考 **Algorithm 3**.

Algorithm 3 Selection

Input: Current population

Output: Selected population

1: Select 10% of the best chromosomes and copy it to the Selected population
2: Generate a random number R from [0,1];
3: **repeat**
4: **if** $Sum_k > R$ **then**
5: Select the kTh chromosome add to the Selected
6: **end if**
7: **until** Size of Population

4.1.3 交叉遗传

在产生新的染色体时，其正确性必须得到担保. 在此问题上，就是指解决方案中的优先级依赖性和任务的独一性必须得到保证. 也就是对一个任务 T_i，需要在 $pred(T_i)$ 和 $succ(T_i)$ 中间的时间执行而不能进行跨越. 于是，在对父串进行处理前，我们可以得到

两个定理：1. 当在一个处理器上删除任务 T_i 时，剩余的任务仍然构成一个正确性的拓扑排序. 2. 当一个处理器两个任务可以作为为 T_i 的前驱和后继时，向这两个任务中间插入一个任务 T_i，该序列仍然是正确的拓扑排序.

于是算法在这里选取了两个现有串进行杂交产生两个子串，这里先在拓扑排序中对不满足以上两个正确性条件的不可交换位点进行标记，在这些位点之外选取一个交换会产生影响的可交换位点对两个串进行交换，将其中一个串的前部分保留，后部分使用另一子串的顺序重新进行拓扑排序，得到的子串显然容易被证实是符合要求的，其中对位点的选取可以利用依赖关系，在满足符合概率的选择机会的同时，前期更倾向于选择遗传变动较大的位点来加速变异，后期选择变动较小的来并避免离群结果的产生. 交叉部分的具体流程参考 **Algorithm 4**.

Algorithm 4 Crossover

Input: Two chromosomes as parents

Output: Two new chromosomes as children

1: Choose randomly crossover point i that can cross
2: copy the left part of genes from father to the son
3: inherit the genes of the mother that do not appear in the left part of father to the right part of the son
4: change the role of parents and do it again
5: replace two new children with two random selected chromosomes in population.

图 6 是一个交叉的样例.

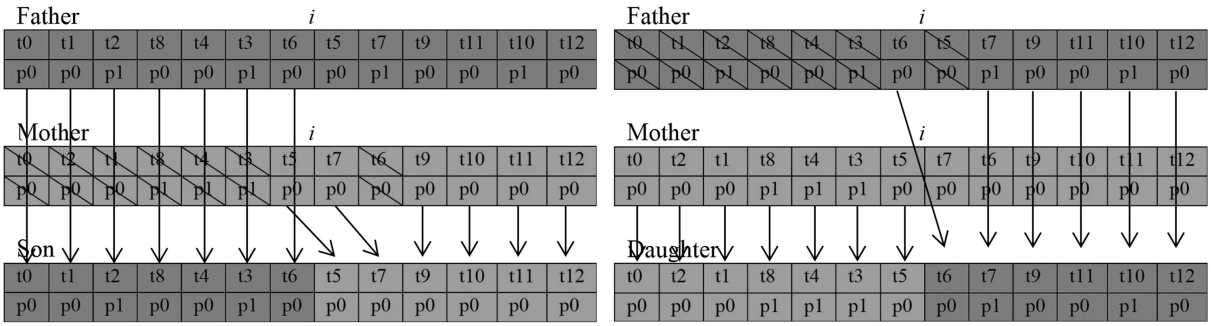


图 6　一个交叉样例

4.1.4 变异算符

在遗传算法中交叉操作可能会因父代种群的限制陷入局部难优化困局，花费大量时间来脱离某一段节点的欠优化方案，算法选择在每次双串交叉后对单串进行变异操作. 在本算法中，调度的拓扑排序依据在设立初始种群时进行的划分操作如**图 5**，分割为多个可突变区域如**图 6**.

t0	t2	t1	t6	t3	t5	t4	t7	t8	t9	t10	t11	t12
p0	p1	p0	p0	p0	p0	p0	p1	p1	p0	p0	p1	p0

图 7　本文实例中的可突变分段

由每一分段中的部分可以并行执行，可以证明其中每个分段内的两个任务发生调换，不会对调度的正确性产生影响. 而同样，可以证明对此拓扑排序，不同分段中的任务如果调换后能够保证其正确性，则将更靠前的分段优先执行是有利于调度的. 则在编译操作中，只需考虑随机选取划分结果中相同分段的任务进行调换. 操作的具体步骤参考 **Algorithm 5**

Algorithm 5 Mutation

Input: A random chosen chromosome

Output: A mutated chromosomes

1: Choose randomly a gene t_i

2: Generate a new child by interchanging gene t_i with a gene in the same segment

4.1.5 反转算符

常规的变异形式可能会消耗很长的时间来获取一个相对优化的结果，本算法中在 **Algorithm 1** 的 **Mutation** 部分后增加了一个新的步骤，采取了一种更为大胆的变异方式对相同分段内的任务进行反转操作. 由 **Algorithm 5** 的正确性，可以知道这样的变异方式是能满足调度的正确性的. 于是对一些段内预测情况不是很良好的基因串，采取反转策略，将该段翻转.

4.2 复杂性分析

根据遗传算法的基本形式，主要的时间花费产生在算法的循环过程中. 而循环中的几个部分，其中筛选、突变、反转和交叉方法的时间复杂度都是 $O(|\mathcal{T}|^2)$，适应程度评估的复杂度为 $O(|E| \times m)$. 设最高迭代次数为 $\mathcal{G}$，则算法总体的复杂度便在 $O(\mathcal{G} \times (3 \times |\mathcal{T}|^2 + |E| \times m))$. 而占用空间为 $O(\mathcal{M} \times |\mathcal{T}| \times 2)$，其中 $\mathcal{M}$ 代表种群大小，对该问题已经相当优秀.

4.3 算法总结

该算法实现了一个非随机的出师任务群体，将初始种群转变为具有一个相对优化的调度的种群，改进遗传功能来确保多样性和一致性，同时减少了在遗传过程中出现重复

的次数，减少执行的时间，更快的获取相对优化的结果，减小了可能存在的局部最优情况对算法正确性的影响.

五、 总结和后记

本文是一篇针对多处理器系统上的具有约束关系的任务调度问题的综述、复现和实验性质的文章，是在本小组三名成员针对该问题阅读了大量论文资料、并挑选其中的几篇做复现和改进实验的基础上撰写而成的. 我们非常清楚地认识到，这个问题是一个经典而“古老”的问题，并不是近年来科研前沿的热点. 我们之所以选这个问题作为研究对象，有三点原因：其一，该问题是经典的强 NP−Hard 问题，是《算法设计与分析》课程中“双机调度”、“多机调度”问题的升级版本，与课内知识有一定的联系性，同时又聚焦于算法本身，其背后的思想可以拓展到算法设计的广阔领域. 其二，该问题的研究脉络非常有特点，前期基本上都在设计传统启发式算法，而后期又是清一色的设计遗传算法，因此我们认为此问题是一个在力所能及范围之内的较为有趣的问题. 其三，由于前期课程《人工智能引论》中，授课老师曾简单提及过遗传算法，却并没有给出实例，因此我们希望自己探索和实现一个遗传算法，并通过实验分析其效果和改进方案.

通过本次小组项目的实践，本小组三名成员查找和阅读英文文献、梳理和总结文献内容、分析和复现算法以及项目报告写作的能力都得到了极大的提升，这也是我们最大的收获.

本研究报告是本小组三名成员共同完成的，报告撰写的原则依据的是《算分第 1 次小班课.pptx》第 6 页中“要讲清楚调研的问题与解决方案，让读者可以零基础看懂你想说什么”的要求，因此力求完整、详细、清晰、准确. 其中，第一部分和第三部分主要由陈奕阳同学完成，第二部分主要由邵子建同学完成，第四部分主要由张皓禹同学完成.

参考文献

[1] Y. Robert, “Task graph scheduling,” in Encyclopedia of Parallel Computing (D. Padua, ed.), Springer, 2011.

[2] T. C. Hu, “Parallel sequencing and assembly line problems,” Operations Research, vol. 9, no. 6, pp. 841–848, 1961.

[3] H. Topcuoglu, S. Hariri, and M.-Y. Wu, “Task scheduling algorithms for heterogeneous processors,” Proceedings of the Heterogeneous Computing Workshop, pp. 3–14, 1999.

[4] T. Yang and A. Gerasoulis, “Scheduling parallel tasks on an unbounded number of processors,” IEEE Transactions on Parallel and Distributed Systems, vol. 5, no. 9, pp. 951–967, 1994.

[5] I. Ahmad and Y.-K. Kwok, “On exploiting task duplication in parallel program scheduling,” IEEE Transactions on Parallel and Distributed Systems, vol. 9, no. 9, pp. 872–892, 1998.

[6] E. Hou, N. Ansari, and H. Ren, “A genetic algorithm for multiprocessor scheduling,” IEEE Transactions on Parallel and Distributed Systems, vol. 5, no. 2, pp. 113–120, 1994.

[7] M. Akbari, H. Rashidi, and S. H.Alizabeh, “An enhanced genetic algorithm with new operators for task scheduling in heterogeneous computing systems,” Engineering Applications of Artificial Intelligence, vol. 61, pp. 35–46, 2017.

[8] E. C. da Silva and P. H. R. Gabriel, “A comprehensive review of evolutionary algorithms for multiprocessor dag scheduling,” Computation, vol. 8, no. 26, pp. 1–16, 2020.

[9] S. Narita and H. Kasahara, “Practical multiprocessor scheduling algorithms for efficient parallel processing,” IEEE Transactions on Computers, vol. 33, no. 11, pp. 1023–1029, 1984.

[10] A. Y. Zomaya, C. Ward, and B. Macey, “Genetic scheduling for parallel processor systems: Comparative studies and performance issues,” IEEE Transactions on Parallel and Distributed Systems, vol. 10, no. 8, pp. 795–812, 1999.

[11] A. S. Wu, H. Yu, S. Jin, K.-C. Lin, and G. Schiavone, “An incremental genetic algorithm approach to multiprocessor scheduling,” IEEE Transactions on Parallel and Distributed Systems, vol. 15, no. 9, pp. 824–834, 2004.

[12] Y. Kwok and I. Ahmad, "Benchmarking the task graph scheduling algorithms," in Parallel Processing Symposium, International, (Los Alamitos, CA, USA), p. 0531, IEEE Computer Society, apr 1998.

[13] Kasahara and Narita, "Practical multiprocessor scheduling algorithms for efficient parallel processing," IEEE Transactions on Computers, vol. C-33, no. 11, pp. 1023–1029, 1984.

[14] L. Wang, H. J. Siegel, V. P. Roychowdhury, and A. A. Maciejewski, "Task matching and scheduling in heterogeneous computing environments using a genetic-algorithm-based approach," Journal of Parallel and Distributed Computing, vol. 47, no. 1, pp. 8–22, 1997.

[15] R. Correa, A. Ferreira, and P. Rebreyend, "Scheduling multiprocessor tasks with genetic algorithms," IEEE Transactions on Parallel and Distributed Systems, vol. 10, no. 8, pp. 825–837, 1999.

[16] Y. Xu, K. Li, J. Hu, and K. Li, "A genetic algorithm for task scheduling on heterogeneous computing systems using multiple priority queues," Information Sciences, vol. 270, pp. 255–287, 2014.

[17] X. Wang, C. S. Yeo, R. Buyya, and J. Su, "Optimizing the makespan and reliability for workflow applications with reputation and a look-ahead genetic algorithm," Future Generation Computer Systems, vol. 27, no. 8, pp. 1124–1134, 2011.

[18] F. A. Omara and M. M. Arafa, "Genetic algorithms for task scheduling problem," Journal of Parallel and Distributed Computing, vol. 70, no. 1, pp. 13–22, 2010.