2025

汇报人：1024040801 白泽楠

目 录

CONTENTS

01 问题描述

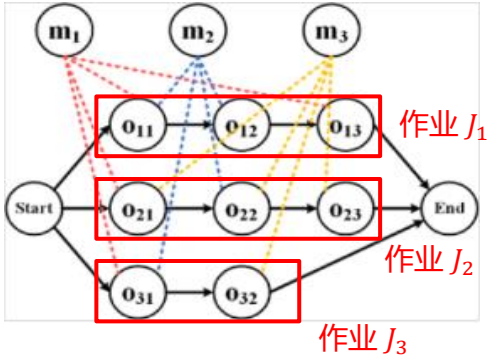
02 求解框架

03 方法 & 实现

问题描述

任务模型

- **作业**：$\mathcal{J} = \{J_1, J_2, \ldots, J_n\}$
- **操作**：作业 J_i 被分解成若干操作
  - 用 $\boldsymbol{O_{ij}}$ 表示第 i 个作业的第 j 个操作
  - 这些操作**必须按照特定先后顺序**处理



资源模型

- **机器**：$\mathcal{M} = \{M_1, M_2, \ldots, M_m\}$
  - 每台机器在**同一时刻**只能处理**一个**操作
- $\mathcal{M}_{ij}$：可处理操作 O_{ij} 的的机器构成的集合，$\mathcal{M}_{ij} \subseteq \mathcal{M}$
- p_{ijk}：操作 O_{ij} 由机器 $M_k \in \mathcal{M}_{ij}$ 进行处理，所需的时间

问题描述

调度模型

- **任务角度**：对于操作 O_{ij}
  - 确定 O_{ij} **分配**给哪一个机器处理
  - 确定 O_{ij} 的**起始时间** S_{ij} 、**完成时间** C_{ij}
- **资源角度**：对于机器 $M_k \in \mathcal{M}$
  - 确定 M_k 何时被哪一个作业的第几个操作占用，何时空闲

约束

- $O_{i,j+1}$ 的开始时间不能早于 O_{ij} 的完成时间：

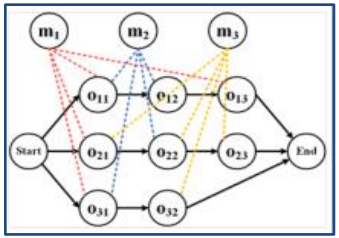
$$S_{i,j+1} \geq C_{ij}, \qquad i = 1,2,\dots,n$$

- 每个操作都必须完成且只能完成一次
- **操作**在机器上处理时**不能被抢占**
- 每个**机器**在同一时刻**只能处理一个操作**

优化目标：**最小化**最大完成时间 $\min\{C_{\max}\}$

求解框架

- **顶层算法**
  - **目标**：生成合法的操作处理序列，实现**操作选择**
  - **方法**：遗传算法、模拟退火等启发式算法
- **底层算法**
  - **目标**：按照某种（或某些）预设的机器分配策略，将操作依次分配到合适的机器上，实现**机器选择**
  - **策略**：
    - ✓ 最早完成机器优先
    - ✓ 最早开始机器优先
    - ✓ 最小 gap 机器优先
    - ✓ 最小 gap 机器优先，其次最早完成机器优先
    - ✓ 最早完成机器优先，其次最小 gap 机器优先



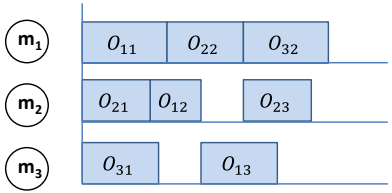
FJSP 实例

顶层算法

O_{11}	O_{21}	O_{31}	O_{12}	O_{22}	O_{32}	O_{13}	O_{23}
O_{21}	O_{11}	O_{22}	O_{12}	O_{31}	O_{32}	O_{13}	O_{23}
⋮							
O_{31}	O_{32}	O_{11}	O_{12}	O_{21}	O_{22}	O_{13}	O_{23}

合法操作序列

底层算法



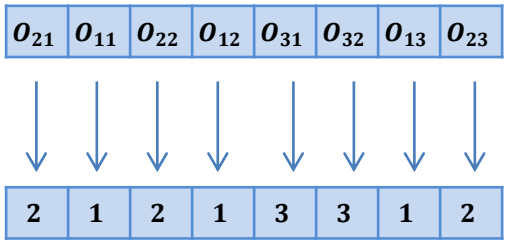
调度方案

方法 & 实现　遗传算法

➢ 遗传算法

- **染色体**：FJSP 实例的某一合法的**操作序列**，应满足操作的**拓扑序**
- **种群**：由多个染色体组成的集合
- **适应度**：最大完成时间 $C_{\max}$ 越小，适应度越大，第 i 个染色体的适应度：

$$① \ f_i = \frac{1}{C_{\max}^{(i)}} \quad 或 \quad ② \ f_i = \frac{1}{C_{max}^{(i)} - C_{max} + 1}$$



操作序列染色体编码

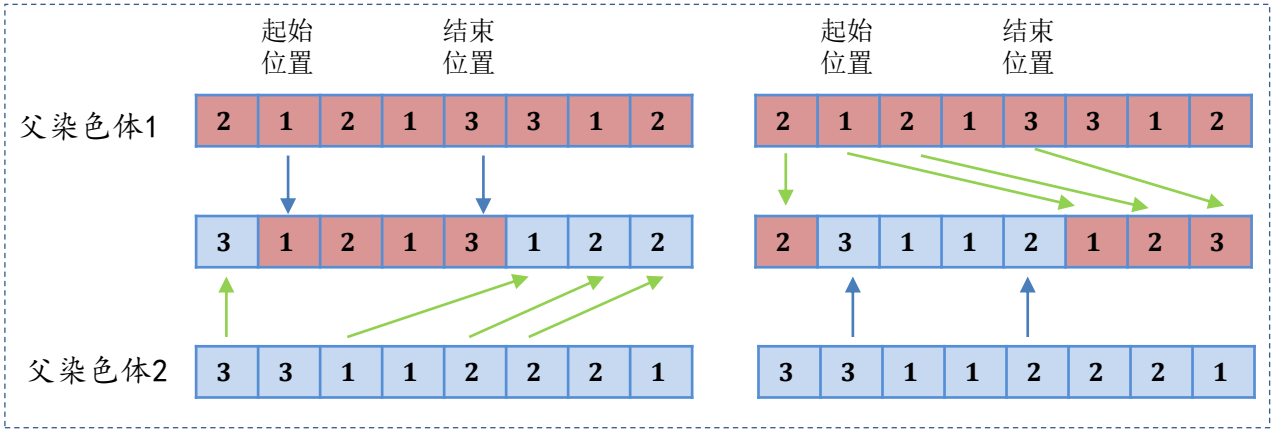
	染色体1	染色体2	染色体3
最大完成时间	83	84	85
适应度①	$\frac{1}{83} \approx 0.012$	$\frac{1}{84} \approx 0.012$	$\frac{1}{85} \approx 0.012$
适应度②	$\frac{1}{83-83+1} = 1$	$\frac{1}{84-83+1} = 0.5$	$\frac{1}{85-83+1} \approx 0.333$

第 ② 种适应度计算方式增强了染色体的区分度，更有利于轮盘赌选择更优的个体

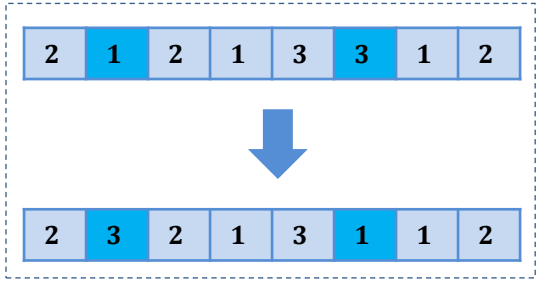
方法 & 实现　遗传算法

➢ 遗传算法

- **选择**：轮盘赌选择，第 i 个染色体被选中的概率 $p_i = \frac{f_i}{\sum_{j=1}^{n} f_j}$
- **交叉**：顺序交叉，从父染色体中随机选择起始位置和结束位置，子代保留父染色体起始和结束位置之间的基因，其余位置用使用另一个父染色体的基因填充
- **变异**：随机交换染色体中两个位置的基因



交叉操作



变异操作

方法 & 实现　遗传算法

```python
class GASolver:
    def __init__(self, fjsp_data, population_size, generations,
                 crossover_rate, mutation_rate):
        '''
        fjsp_data:  FJSP调度问题实例，是一个 FjspInstance 类
        population_size: 种群大小
        generations: 代数
        crossover_rate: 交叉率
        mutation_rate: 变异率
        '''
        self.solver_name = 'Genetic Algorithm Solver:'

        self.fjsp_data = fjsp_data
        self.population_size = population_size
        self.generations = generations
        self.crossover_rate = crossover_rate
        self.mutation_rate = mutation_rate

        self.operation_num_of_each_job = self.fjsp_data.get_operation_num() # 一个列表
        self.population = self.init_population()
        self.fitness = []

        self.best_makespan = -1
        self.best_Oij_start_time = []
        self.best_Oij_completion_time = []
        self.best_Oij_assignment = []
        self.best_machine_useage_info = []
        self.best_job_seq = []

        # 初始化种群后，更新适应度和最优值
        self.update_fitness_and_answer()
```

- 初始化算法超参数和变量：
  - 种群规模：50
  - 迭代次数：100
  - 交叉率：0.8
  - 变异率：0.05
- 初始化种群

```python
# 初始化种群
def init_population(self):
    job_seq = []
    oper_num_total = 0
    for job_id, oper_num in enumerate(self.operation_num_of_each_job):
        job_seq += [job_id] * oper_num
        oper_num_total += oper_num

    job_seq = np.array(job_seq)     # list 转化为 numpy 数组
    rng = np.random.default_rng()
    return [job_seq[rng.permutation(oper_num_total)] for _ in range(self.population_size)]
```

方法 & 实现　遗传算法

```python
# 选择过程
def selection(self):
    # 轮盘赌选择
    select_probability = self.fitness / self.fitness.sum()
    selected_index = np.random.choice(range(self.population_size),
                                      size=self.population_size, p=select_probability)
    selected_population = [self.population[i] for i in selected_index]
    return selected_population
```

```python
# 交叉过程
def crossover(self, selected_population):
    next_population = []
    for i in range(0, self.population_size, 2):
        if np.random.rand() < self.crossover_rate:
            new_offspring1, new_offspring2 = self.get_new_offspring(selected_population[i],
                                                                    selected_population[i + 1])
            next_population.extend([new_offspring1, new_offspring2])
        else:
            next_population.append(selected_population[i])
            next_population.append(selected_population[i + 1])
    return next_population
```

```python
# 变异过程
def mutation(self, next_population):
    for i in range(self.population_size):
        if np.random.rand() < self.mutation_rate:
            mutation_positions = np.random.choice(range(len(next_population[i])),
                                                  size=2, replace=False)
            a, b = mutation_positions
            next_population[i][a], next_population[i][b] = next_population[i][b], next_population[i][a]
```

```python
# 种群更新后，计算个体适应度，更新最优解
def update_fitness_and_answer(self):
    self.chromo_makespan = np.array([self.fjsp_data.schedule_in_topo_order(seq)[0] for seq in self.population])

    min_makespan_index = self.chromo_makespan.argmin()

    if self.best_makespan == -1 or self.chromo_makespan[min_makespan_index] < self.best_makespan:
        self.best_job_seq = self.population[min_makespan_index]

        self.best_makespan, self.best_Oij_start_time, self.best_Oij_completion_time, self.best_Oij_assignment,\
            self.best_machine_useage_info = self.fjsp_data.schedule_in_topo_order(self.best_job_seq)

    self.fitness = 1.0 / (self.chromo_makespan - self.chromo_makespan[min_makespan_index] + 1)
```

- 轮盘赌选择过程
- 顺序交叉过程
- 变异过程
- 适应度和最优解的更新

方法 & 实现　遗传算法

```python
# 运行遗传算法，寻找最优解
def run_genetic_algorithm(self):
    for iter_count in range(self.generations):
        selected_population = self.selection()
        next_population = self.crossover(selected_population)
        self.mutation(next_population)
        self.population = next_population
        self.update_fitness_and_answer()
        # print("Makespan: ", self.best_makespan)

    print("Makespan: ", self.best_makespan)

    return self.best_makespan, self.best_Oij_start_time, self.best_Oij_completion_time,\
            self.best_Oij_assignment, self.best_machine_useage_info, self.best_job_seq
```

遗传算法执行流程

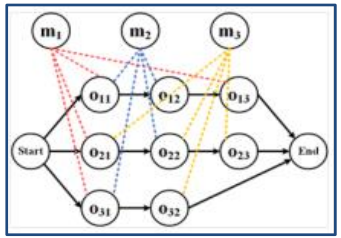
```python
if __name__ == '__main__':
    # 加载 FJSP 实例数据
    fjsp_data = FjspInstance("./data_dev/1005/10j_5m_001.fjs")
    # 初始化求解器
    ga_solver = GASolver(fjsp_data, population_size=50, generations=100, crossover_rate=0.8, mutation_rate=0.05)
    # 执行遗传算法，生成问题的可行解
    ga_solver.run_genetic_algorithm()
    # 检查解的合法性
    fjsp_data.validate_schedule(ga_solver.best_makespan,
                                ga_solver.best_Oij_start_time,
                                ga_solver.best_Oij_completion_time,
                                ga_solver.best_Oij_assignment)
    # 打印 makespan 和调度方案
    show_schedule(ga_solver)
```

- 求解过程：
  - 加载数据
  - 初始化参数和种群
  - 执行遗传算法
  - **检查解的合法性**
  - 打印 makespan 和调度方案

方法 & 实现　模拟退火算法

➢ 模拟退火算法

- **参数**：初始温度 T_0，终止温度 T_f，冷却系数 α，最大迭代次数
- **初始解**：一个特定的合法操作序列
- **邻域解**：随机交换当前解中两个操作的位置，得到邻域解
- **接受准则**：邻域解更优，接受邻域解；否则以 $e^{-\frac{\Delta E}{T}}$ 的概率接受邻域解
- **冷却策略**：每次迭代后，$T \leftarrow \alpha \times T$（$T$：当前温度）
- **终止条件**：当前温度 T 降低到终止温度 T_f，或达到最大迭代次数



O_{11}	O_{12}	O_{13}	O_{21}	O_{22}	O_{23}	O_{31}	O_{32}
1	1	1	2	2	2	3	3

初始解

```python
# 生成初始解
def generate_initial_solution(self):
    operation_num_of_each_job = self.fjsp_data.get_operation_num()
    job_seq = []
    for job_id in range(len(operation_num_of_each_job)):
        job_seq += [job_id] * operation_num_of_each_job[job_id]
    return np.array(job_seq)
```

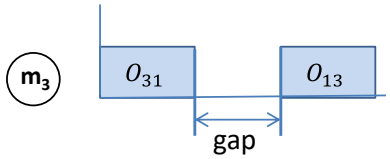
```python
# 生成邻域解（通过随机扰动）
def generate_neighbour(self):
    new_seq = self.job_seq.copy()
    i, j = random.sample(range(len(new_seq)), 2)
    new_seq[i], new_seq[j] =  new_seq[j], new_seq[i]
    return new_seq
```

```python
def  run_sim_anneal(self):
    iter_count = 0
    while iter_count < self.max_iter and self.Tf < self.T:
        new_seq = self.generate_neighbour()
        new_mkspan, new_start_time, new_completion_time, new_assignment,\
            new_machine_useage_info = self.fjsp_data.schedule_in_topo_order(new_seq)

        delta_energy = new_mkspan - self.cur_makespan
        if delta_energy < 0 or random.random() < math.exp(-delta_energy / self.T):
            self.cur_makespan = new_mkspan
            self.job_seq = new_seq
            if new_mkspan < self.best_makespan:
                self.best_makespan = new_mkspan
                self.best_Oij_start_time = new_start_time
                self.best_Oij_completion_time = new_completion_time
                self.best_Oij_assignment = new_assignment
                self.best_machine_useage_info = new_machine_useage_info
                self.best_job_seq = new_seq.copy()

        self.T *= self.cooling_rate
        iter_count += 1

    return self.best_makespan, self.best_Oij_start_time, self.best_Oij_completion_time,\
            self.best_Oij_assignment, self.best_machine_useage_info, self.best_job_seq
```

方法 & 实现　底层算法

➢ 策略

- 最早完成机器优先
- 最早开始机器优先
- 最小 gap 机器优先
- 最小 gap机器优先，其次最早完成机器优先
- 最早完成机器优先，其次最小 gap 机器优先




```
self.strategies_list = {'EARLEST_COMPLETE_TIME_FIRST': 0,
                        'EARLEST_START_TIME_FIRST': 1,
                        'MIN_GAP_FIRST': 2,
                        'MIN_GAP_FIRST_AND_EARLEST_COMPLETE_TIME_SECOND': 3,
                        'EARLEST_COMPLETE_TIME_FIRST_AND_MIN_GAP_SECOND': 4}
```

```
elif strategy == self.strategies_list['EARLEST_COMPLETE_TIME_FIRST']:
    # 最早完成机器优先
    if finish_time < T_end:
        cur_gap = max(left_gap, right_gap)  # 更新 gap 大小
        machine_select_index = machine_id   # 更新选择的机器编号
        T_start = cur_earliest_start_time   # 更新操作的开始时间
        T_end = finish_time                 # 更新操作的结束时间
```

```
elif strategy == self.strategies_list['EARLEST_START_TIME_FIRST']:
    # 最早开始机器优先
    if cur_earliest_start_time < T_start:
        cur_gap = max(left_gap, right_gap)  # 更新 gap 大小
        machine_select_index = machine_id   # 更新选择的机器编号
        T_start = cur_earliest_start_time   # 更新操作的开始时间
        T_end = finish_time                 # 更新操作的结束时间
```

```
elif strategy == self.strategies_list['MIN_GAP_FIRST_AND_EARLEST_COMPLETE_TIME_SECOND']:
    # 最小 gap 优先，其次最早完成优先
    if max(left_gap, right_gap) < cur_gap or (max(left_gap, right_gap) == cur_gap and finish_time < T_end):
        cur_gap = max(left_gap, right_gap)  # 更新 gap 大小
        machine_select_index = machine_id   # 更新选择的机器编号
        T_start = cur_earliest_start_time   # 更新操作的开始时间
        T_end = finish_time                 # 更新操作的结束时间
```

```
elif strategy == self.strategies_list['MIN_GAP_FIRST']:
    # 最小 gap 优先
    if max(left_gap, right_gap) < cur_gap:
        cur_gap = max(left_gap, right_gap)  # 更新 gap 大小
        machine_select_index = machine_id   # 更新选择的机器编号
        T_start = cur_earliest_start_time   # 更新操作的开始时间
        T_end = finish_time                 # 更新操作的结束时间
```

```
elif strategy == self.strategies_list['EARLEST_COMPLETE_TIME_FIRST_AND_MIN_GAP_SECOND']:
    # 最早完成优先，其次最小 gap 优先
    if finish_time < T_end or (finish_time == T_end and max(left_gap, right_gap) < cur_gap):
        cur_gap = max(left_gap, right_gap)  # 更新 gap 大小
        machine_select_index = machine_id   # 更新选择的机器编号
        T_start = cur_earliest_start_time   # 更新操作的开始时间
        T_end = finish_time                 # 更新操作的结束时间
```

方法 & 实现　结果

不同算法求解相同 FJSP 实例的结果

	DRL-S (采样策略)	DRL-G (贪婪策略)	SA (模拟退火)	GA (遗传算法)
./1005/10j_5m_001.fjs	85	86	85	**82**
./1005/10j_5m_002.fjs	88	89	**87**	**85**
./1510/15j_10m_001.fjs	150	155	150	**149**
./1510/15j_10m_002.fjs	163	163	167	**159**
./2005/20j_5m_001.fjs	213	217	**204**	**205**
./2005/20j_5m_002.fjs	198	206	**190**	**188**
./2010/20j_10m_001.fjs	215	213	214	**208**
./2010/20j_10m_002.fjs	191	192	**189**	**189**

结论:

对于 8 个不同的 FJSP 实例，GA 在每个实例上的表现均优于 DRL 方法；

SA 在大部分实例上的表现不劣于 DRL 方法

注:

1. SA 和 GA 的底层算法：最早完成机器优先，其次最小 gap 机器优先
2. 此处的遗传算法采用公式 $f_i = \frac{1}{C_{\max}^i - C_{\max}+1}$ 计算个体适应度

方法 & 实现　结果

注:
1. 顶层算法：遗传算法
2. 此处的遗传算法采用公式 $f_i = \frac{1}{C_{\max}^i}$ 计算个体适应度

不同底层算法求解结果比较

	DRL-S	DRL-G	最早完成优先	最早开始时间优先	最小gap优先	最小gap优先，其次最早完成	最早完成优先，其次最小gap
./1005/10j_5m_001.fjs	85	86	85	87	213	**84**	**84**
./1005/10j_5m_002.fjs	88	89	**87**	89	160	89	**86**
./1510/15j_10m_001.fjs	150	155	156	167	492	178	153
./1510/15j_10m_002.fjs	163	163	170	179	673	179	165
./2005/20j_5m_001.fjs	213	217	**211**	214	416	**211**	**207**
./2005/20j_5m_002.fjs	198	206	**188**	**196**	320	**195**	**188**
./2010/20j_10m_001.fjs	215	213	**208**	226	966	229	**209**
./2010/20j_10m_002.fjs	191	192	192	208	838	205	195

表现： 策略5 ≥(≈) 策略1 > 策略4 ≈ 策略2 ≫ 策略3

方法 & 实现　AI算法构想

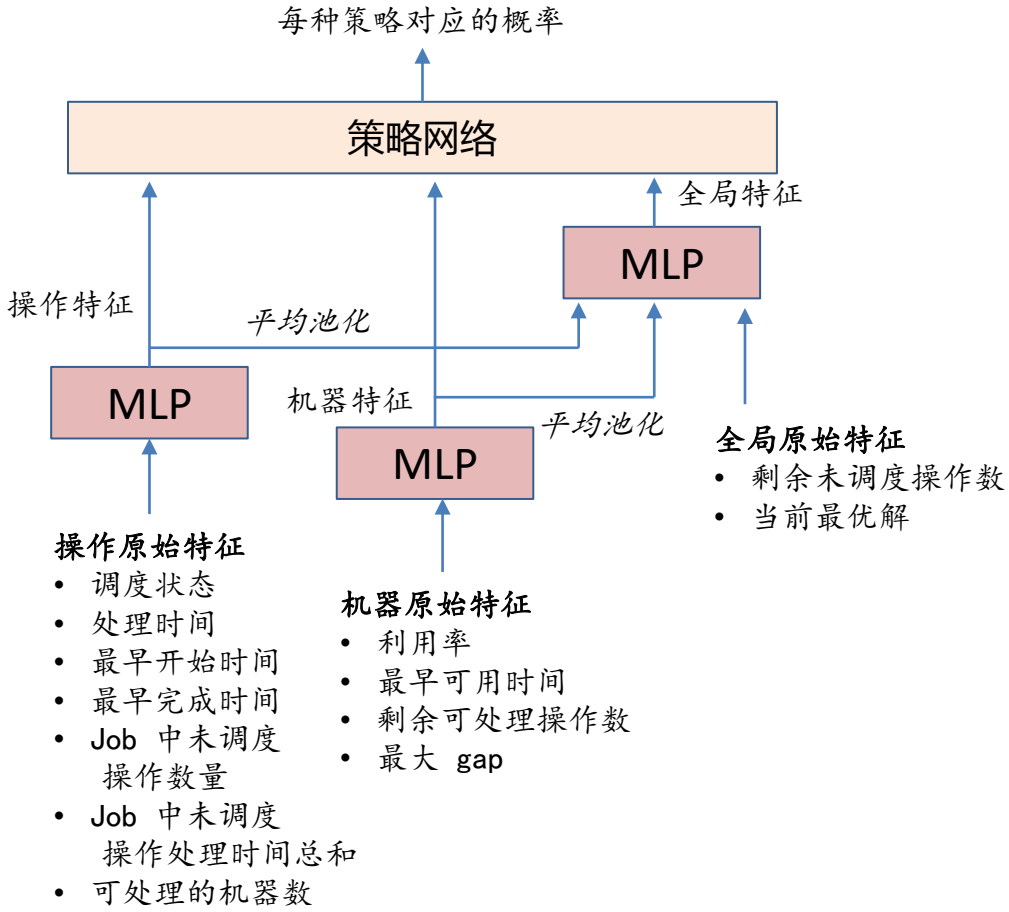
局限性

- 前述方法底层采用**单一**的策略，不能充分发挥每种策略的优势，不够灵活
- 最早完成机器优先策略在每一步做出局部最优的选择，最终未必能达到全局最优

改进思路

- 保留顶层算法，仍然使用 SA 或 GA 生成合法操作序列，实现**操作选择**
- 改进底层算法，引入**策略选择**机制（确定了策略，策略将自动为操作选择合适的机器）
  - ✓ 训练一个神经网络模型，以调度状态（选择的操作、机器状态、全局信息）作为输入，预测应当选择的策略
  - ✓ 调度状态：考虑使用多个 MLP 分别从操作、机器、全局信息中进行特征提取，拼接起来
  - ✓ 训练方式：Actor-Critic 算法 / PPO 算法

方法 & 实现　AI算法构想


每种策略对应的概率
策略网络
全局特征
MLP
操作特征
平均池化
MLP
机器特征
平均池化
MLP
全局原始特征
• 剩余未调度操作数
• 当前最优解
操作原始特征
• 调度状态
• 处理时间
• 最早开始时间
• 最早完成时间
• Job 中未调度操作数量
• Job 中未调度操作处理时间总和
• 可处理的机器数
机器原始特征
• 利用率
• 最早可用时间
• 剩余可处理操作数
• 最大 gap

THANK YOU