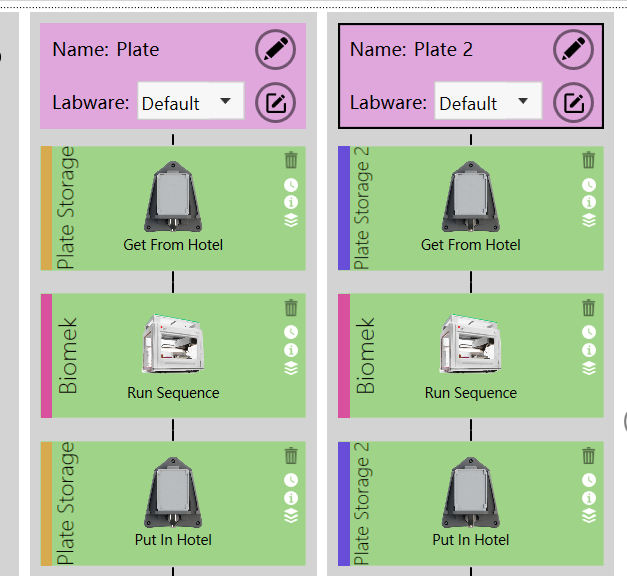
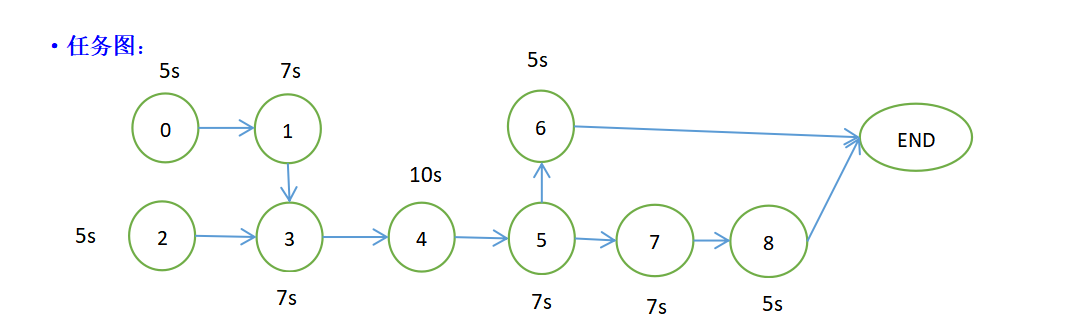
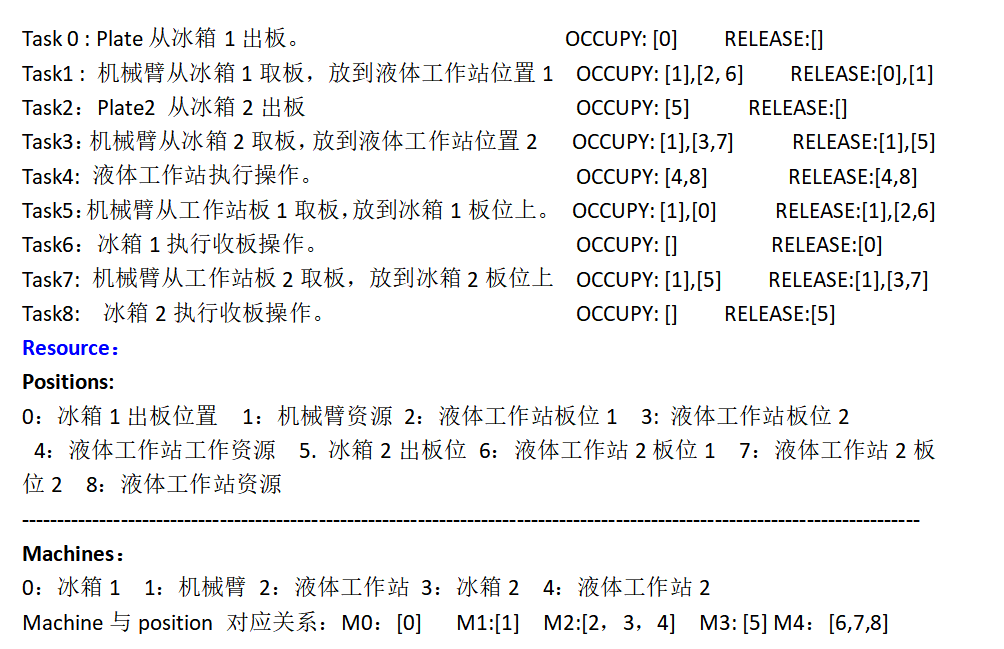
**算法输入说明文档**

|  |
| --- |
| 1. 任务图说明 |
| 1. 基本数据结构说明 |
| 1. 算法输入样例 |

1. **任务图说明**

下图为：两个泳道内存在多板设备池的情况。







任务图详细信息可在上述test-cast-7例子中查看。

算法的输入为任务图的形式进行输入，其中任务图主要包括两个主要部分：**任务相关建模、资源相关建模**。下面分别从**资源建模**和**任务建模**两个部分进行说明。

1. **资源建模：**

**· Machines表示当前可用设备集合，对应于project数据结构中的instrument部分内容，表示所有可用的设备。**

**· Positions是所有任务分配资源的基本单位。分为两种类型，一种类型是设备的工作资源，另一种类型是设备的板位资源。设备通常对应一至多个板位资源与一个工作资源。**

**具体来说：机械臂仅含一个工作资源、单板设备包含一个工作资源和一个板位资源**

**多板设备包含一个工作资源和多个板位资源。**

**· Position与Machine对应关系：表示基础资源单位与设备之间的对应关系，例 如一个设备可能包含多个基础资源单位。**

1. **任务建模**

**·任务类型：任务图中任务分为两种类型：节点设备工作任务、机械臂转移任务。**

**·任务间拓扑关系：除液体工作站类型外，任务之间关系按照泳道上的前驱后继进行设置。若存在液体工作站，液体工作站的sequence中每个step作为一个任务。每个step对应于泳道中的节点。**

**·任务占用设备与释放设备：OCCUPY，RELEASE**

**占用设备：占用position的ID，根据工艺类型判断该任务需要什么类型的工作资源或者位置资源。例如工作任务一般需要占用设备工作资源**

**释放设备：任务执行会释放的设备资源。例如机械臂从A转移至B，则会释放A的转移资源。**

**例子：OCCUPY = [1],[2,7] 表示需要占用资源的id 为1，另外2和7中选择一个即可。**

**·任务执行时间：表示该任务的执行时间。对于工作任务对应于estDuration，对 于转移任务与机械臂时间相关。具体依据工艺类型进行判断。**

**·任务约束关系：有两种关系，一种是占用约束。另一种是释放约束。主要是对 任务占用、释放设备有多个候选时的约束。**

**占用约束：task[2].occ\_dependency[0] = 1 表示任务2的第0个要求资源需要和任务1占用的资源相同。 这里是由于考虑液体工作站一个序列转换成的任务，其占用的资源必须属于同一个液体工作站。**

**释放约束：task[2].rel\_dependency[0] = 1 表示任务2的第0个释放资源时要选择任务1占用的资源相同的资源。**

1. **基本数据结构说明**

算法输入需求为tasks[]数组，positions[]数组，machines[]数组的初始化内容，数组中内容为对应的类。三个部分能够表示任务图建模与资源建模内容。

以下三个类的内容。Task、Position、Machine

1. **Task**

class Task:

    """

    Base Task job.

    """

    def \_\_init\_\_(self, id, occupy) -> None:

        self.id = id

        # self.occupy表示占用的设备资源，例如 [1,2,3] 表示从1，2，3中选择一个

        self.occupy = occupy

        self.release = []

        self.occupy\_dependency = []

        self.release\_dependency = []

        self.dependency2 = -1

        # self.status表示该任务当前状态，0表示未执行，1表示已执行

        self.status = 0

        self.priority = 0

        self.time = 0

        self.pre = None

        self.next = None

        self.machine = -1  # machine 表示当前task在哪个machine上工作

        self.position = -1  # position 表示当前task在哪个位置资源上工作

        self.release\_machine = []  # 表示当前task释放的machine

        self.release\_position = [] # 表示当前task释放的position

        self.dependency = -1  # dependency 表示当前task和哪个task需要有相同的设备

        self.available = 0

        self.start\_time = 0

        self.heuristic = 0

    def \_\_lt\_\_(self, other):

        """定义<比较操作符"""

        if self.priority == other.priority:

            return self.heuristic > other.heuristic

        return self.priority > other.priority

以下内容需要给出任务的数量，并完成每个任务的初始化。

tasks = [Task(t, []) for t in range(args.task\*args.base\_num)]

初始化内容需给出以下内容：

1. **任务图中任务占用和释放的position列表。**

# 设置每个task需要的资源,暂时手动设置一个例子，测试算法是否可行，后面会从数据结构中生成下面的内容。

    tasks[0].occupy = [[0]]  # 0：冰箱1出Plate

    tasks[0].release = []

1. **任务拓扑关系**

  tasks[0].pre = []

  tasks[1].pre = [0]

tasks[0].next = [1]

tasks[1].next = [3]

1. **任务时间**

tasks[0].time = 5

    tasks[1].time = 7

1. **任务间约束**

tasks[3].occupy\_dependency[1] = 1

    tasks[5].release\_dependency[1] = 1

    tasks[6].release\_dependency[1] = 1

    tasks[10].occupy\_dependency[1] = 9

    tasks[12].release\_dependency[1] = 9

    tasks[13].release\_dependency[1] = 9

1. **Position**

class Position:

    def \_\_init\_\_(self, id) -> None:

        self.id = id

        self.status = 0  # status为0表示空闲、为1表示忙碌。

        self.machine = None

1. **Machine**

class MachineBase:

    def \_\_init\_\_(self, id) -> None:

        self.id = id

1. **算法输入样例**

**对应于上述任务图部分。**

tasks = [Task(t, []) for t in range(args.task\*args.base\_num)]

    base\_num = args.base\_num

    positions = [Position(p) for p in range(args.position)]

    heuristics = args.heuristics

    # 设置每个task需要的资源,暂时手动设置一个例子，测试算法是否可行，后面会从数据结构中生成下面的内容。

    tasks[0].occupy = [[0]]  # 0：冰箱1出Plate

    tasks[0].release = []

    tasks[1].occupy = [[1], [6]]

    tasks[1].release = [[0], [1]]

    tasks[2].occupy = [[9]]  #

    tasks[2].release = []

    tasks[3].occupy = [[1], [7]]  #

    tasks[3].release = [[1], [9]]

    tasks[4].occupy = [[8]]  #

    tasks[4].release = [[8]]

    tasks[5].occupy = [[1], [2]]  #

    tasks[5].release = [[1], [6]]

    tasks[6].occupy = [[1],[4]]  #

    tasks[6].release = [[1],[7]]

    tasks[7].occupy = [[3]]  # 0：冰箱1出Plate

    tasks[7].release = [[3]]

    tasks[8].occupy = [[5]]

    tasks[8].release = [[5]]

    tasks[9].occupy = [[1], [6]]  #

    tasks[9].release = [[1], [2]]

    tasks[10].occupy = [[1],[7]]  #

    tasks[10].release = [[1],[4]]

    tasks[11].occupy = [[8]]  # 0：冰箱1出Plate

    tasks[11].release = [[8]]

    tasks[12].occupy = [[1],[0]]

    tasks[12].release = [[1],[6]]

    tasks[13].occupy = [[1],[9]]  #

    tasks[13].release = [[1],[7]]

    tasks[14].occupy = []  # 0：冰箱1出Plate

    tasks[14].release = [[0]]

    tasks[15].occupy = []

tasks[15].release = [[9]]

# 构建TaskGraph，连接task之间的有向图，每个task维护其前驱task集合。 有向图使用前驱进行表示，便于查找任务的前驱任务是否完成。

    tasks[0].pre = []

    tasks[1].pre = [0]

    tasks[2].pre = []

    tasks[3].pre = [1,2]

    tasks[4].pre = [3]

    tasks[5].pre = [4]

    tasks[6].pre = [5]

    tasks[7].pre = [5]

    tasks[8].pre = [6]

    tasks[9].pre = [7]

    tasks[10].pre = [8,9]

    tasks[11].pre = [10]

    tasks[12].pre = [11]

    tasks[13].pre = [12]

    tasks[14].pre = [12]

    tasks[15].pre = [13]

    tasks[0].next = [1]

    tasks[1].next = [3]

    tasks[2].next = [3]

    tasks[3].next = [4]

    tasks[4].next = [5]

    tasks[5].next = [6,7]

    tasks[6].next = [8]

    tasks[7].next = [9]

    tasks[8].next = [10]

    tasks[9].next = [10]

    tasks[10].next = [11]

    tasks[11].next = [12]

    tasks[12].next = [13,14]

    tasks[13].next = [15]

    tasks[14].next = []

    tasks[15].next = []

machines = [MachineBase(m+1) for m in range(args.machine)]

    tasks[0].time = 5

    tasks[1].time = 7

    tasks[2].time = 5

    tasks[3].time = 7

    tasks[4].time = 10

    tasks[5].time = 7

    tasks[6].time = 7

    tasks[7].time = 10

    tasks[8].time = 10

    tasks[9].time = 7

    tasks[10].time = 7

    tasks[11].time = 10

    tasks[12].time = 7

    tasks[13].time = 7

    tasks[14].time = 5

    tasks[15].time = 5

# 对资源的建模

    positions[0].machine = 0

    positions[1].machine = 1

    positions[2].machine = 2

    positions[3].machine = 2

    positions[4].machine = 3

    positions[5].machine = 3

    positions[6].machine = 4

    positions[7].machine = 4

    positions[8].machine = 4

    positions[9].machine = 5

**算法输入目标需求：依据工艺流程项目的json文件，生成tasks数组，positions数组，machines数组。**

**Positions**

**0：PlateLidOpenAndClose\_1 Position 1**

**1：PlateLidOpenAndClose\_1 Position 2**

**2：PlateLidOpenAndClose\_1 Position 3**

**3：PlateLidOpenAndClose\_1 Position 4**

**4：Victor Nivo\_1**

**5：Janus\_1 Position 1**

**6：Janus\_1 Position 2**

**7：Wellwash\_1**

**8：DENSO RC8\_1**

**9：StorageCarousel\_1**

**10：Multidrop Combi\_1**

**11：PlateLidOpenAndClose\_1 work**

**12：Victor Nivo\_1 work**

**13：Janus\_1 work**

**14：Wellwash\_1 work**

**15：DENSO RC8\_1 work**

**16：StorageCarousel\_1 work**

**17：Multidrop Combi\_1 work**

需要添加一个逻辑，即所有任务规划好后，将任务从任务图后方向前遍历，观察是否有任务能够向后移动，如果可以晚一些执行，那么就尽量晚一些执行。来保证执行的连续性。

计算出所有task的start\_time 和available time后进行一个预处理。

需要维护一个每个资源忙碌的时间段。Step1 要知道每个资源什么时候忙碌什么时候空闲。

当给新任务安排时间时，

思路梳理：

Resultsoutput 模块：新增功能

1. ~~按顺序遍历所有任务，根据任务开始时间和结束时间计算资源占用的时间段。 同时记录该任务与其对应的时间段之间的对应关系。~~
2. ~~对资源占用的时间段根据开始时间进行排序。~~
3. ~~计算每一个板与哪些任务关联。得到一个list~~
4. ~~从最后一块板到第一块倒序遍历。每块板从最后一个任务到前面的任务倒序遍历~~
5. ~~遍历中每个任务，查询其后续任务的最早开始时间。~~
6. ~~查询该任务需要的资源可用时间，在最早开始时间前是否有其他可用时间，如果可以向后推迟则向后推迟~~。对资源时间段占用进行更新。

[10 ,20] [40, 50] [100,200] 有一些忙碌时间段。

想知道时间在20之后 90 之前。长度为10的最靠右的空闲区间，

例如上述区间就应该是 [80, 90]