主题控制逻辑

- 主目录
  - controller.py 主要的控制程序，负责调用核心规划模块和与模拟器交互
  - mapf_lns2_solver.py 与核心规划模块交互的接口
  - c_modules 高效率实现的cython代码，包含了核心规划模块
    - mapf_lns2.pyx 核心规划模块的主要控制器
    - path_table.pyx 负责添加路径（可用于添加路径来防止其他agv与路径相交）
    - sipp.pyx 单路径规划模块

算法

- 可以参考主目录下的 “paper.pdf” 核心规划模块是在这一算法的基础上修改得到的

- 核心思路如下：
  - 首先得到一个初始解
  - 之后每次选择部分agv，为他们重新规划解
  - 多次迭代上述过程，逐步更新解的质量

对于初始已经存在的路径加他们加入path_table成为必须要避免的路径

更新初始的碰撞信息
CG[i, j]=1 表示AGV-i和AGV-j有碰撞

得到当前轮次需要重新规划路径的AGV组，使用get_neighbor来得到
如果是得到初始解（it=0），那么对所有的AGV都要重新规划路径

这一行是对选出来的AGV组进行重规划

如果重规划后得到了更好的解就采用，否则舍弃这一次的结果

```
for i in range(self.AGV):
    start_path = [(-1, start_time[i], start_time[i])] if start_edges[i] == -1 else [(start_edges[i], 0, start_time[i])]
    self.agv_path_id[i] = self.path_table.add_path_to_buffer(
        agent_id=i,
        start_vertex=start_vertices[i],
        path=start_path,
        graph=self.graph
    )
    self.path_table.insert_path(self.agv_path_id[i])
cdef int j
cdef int[:, :] CG_view = self.CG
for i in range(self.AGV):
    for j in range(self.AGV):
        if i >= j:
            CG_view[i, j] = CG_view[j, i]
            continue
        CG_view[i, j] = self.path_table.count_path_pair_collision(self.agv_path_id[i], self.agv_path_id[j]) > 0

cdef int it = 0
cdef double mapf_start_time = time.time()
cdef dpair best_loss = dpair(2.0 * self.AGV * self.AGV, 0)
while it < self.max_replan_steps and time.time() - mapf_start_time < self.max_replan_time:
    # Sample a neighborhood to replan
    if it == 0:
        neighbor = self.get_neighbor("uniform-random", self.AGV)
    else:
        select_command = self.query_next_neighbor_type()
        neighbor = self.get_neighbor(select_command, self.mapf_neighbor_size)
        if neighbor.size() <= 1:
            it += 1 # avoid infinite loop
            continue

    # Replan using sipp
    new_CG, new_path_id = self.replan_iteration(neighbor, start_vertices, start_edges, goal_vertices, start_endpoints, start_time, target_stay_time)

    # Updarte lns and the best solution
    cur_loss = self.evaluate_solution(new_CG, new_path_id)
    if it > 0:
        self.lns_update(select_command, best_loss.first - cur_loss.first, best_loss.second - cur_loss.second)
    if it == 0 or cur_loss < best_loss:
        best_loss = cur_loss
        self.agv_path_id = new_path_id
        self.CG = new_CG.copy()
    else:
        for i in neighbor:
            self.path_table.delete_path(new_path_id[i])
            self.path_table.insert_path(self.agv_path_id[i])
    it += neighbor.size()

return self.CG, [self.path_table.get_path(self.agv_path_id[i])[2] for i in range(self.AGV)]
```

对一组AGV进行重规划

首先将他们的路径全部从path_table中删掉；这时path_table中只剩下非当前AGV组的所有路径

依次按照顺序为当前AGV组的AGV规划路径；规划路径使用的是sipp.pyx中的算法（在sipp.pyx的plan函数），整体思路是找到在与path_table中存在路径的碰撞最少的条件下，一条最短的路径
每次对一个AGV找到路径后，加入path_table中

更新得到当前解的碰撞情况

```
cdef replan_iteration(self, ivec& neighbor, int[:] start_vertices, int[:] start_edges, int[:
    new_path_id = self.agv_path_id
    for i in neighbor:
        self.path_table.delete_path(self.agv_path_id[i])
        new_path_id[i] = -1
    new_CG = self.CG.copy()
    cdef int[:, :] new_CG_view
    new_CG_view = new_CG
    for i in neighbor:
        p = self.sipp.plan(
            start=start_endpoints[i],
            target=goal_vertices[i],
            graph=self.graph,
            path_table=self.path_table,
            start_time=start_time[i],
            target_stay_time=target_stay_time[i],
        )
        if start_edges[i] != -1:
            p = [(start_edges[i], 0.0, start_time[i])] + p
        new_path_id[i] = self.path_table.add_path_to_buffer(
            agent_id=i,
            start_vertex=start_vertices[i],
            path=p,
            graph=self.graph
        )
        collision_nodes = self.path_table.count_collision_on_path(i, new_path_id[i]).second
        for j in range(self.AGV):
            if i == j or new_path_id[j] == -1:
                continue
            new_CG_view[i, j] = new_CG_view[j, i] = 0
        for collision_node in collision_nodes:
            j = collision_node.agent_id
            if i == j or new_path_id[j] == -1:
                continue
            new_CG_view[i, j] = new_CG_view[j, i] = 1
        self.path_table.insert_path(new_path_id[i])
    return new_CG, new_path_id
```

如何选择重规划的AGV组

选择AGV组是通过get_neighbor函数实现的，其中select_command会根据当前解的质量动态选择使用哪一个选择方法；注意get_neighbor得到的AGV组的优劣只会轻微影响解的好坏，并且这一部分可以根据业务需要进行专门设计

一共有六种不同的选择方法；前三种是对应当前有碰撞的情况用于优先解决碰撞，后三种则是在无碰撞的情况下优先提升解的质量（总时长）

非纯随机的斯种选择方法的思想如下：

- “collision-based” 在碰撞关系图上找一个连通块，在连通块上随机游走将没碰到过的AGV依次加入；如果还不够，则每次随机一个已经加入的AGV的路径，从路径上的某一点开始随机游走，将碰到的AGV加入
- “target-based” 先随机找一个AGV，然后找到那些经过该AGV起点和终点的AGV然后逐步加入；寻找占用该AGV起点终点的方法也有一些类似随机游走
- “agent-based” 找到延迟最大的AGV，在它的路径上随机游走找到阻碍他缩短时间的AGV加入
- “map-based” 随机寻找一个点，从这个点开始向周围拓展加入所有碰到的AGV

```
select_command = self.query_next_neighbor_type()
neighbor = self.get_neighbor(select_command, self.mapf_neighbor_size)
```

```
cdef ivec get_neighbor(self, str select_command, size_t neighbor_size):
    cdef ivec neighbor
    if select_command == "collision-based":
        neighbor = self.generate_collision_based_neighbor(neighbor_size)
    elif select_command == "target-based":
        neighbor = self.generate_target_based_neighbor(neighbor_size)
    elif select_command == "conflict-random":
        neighbor = self.generate_conflict_random_neighbor(neighbor_size)
    elif select_command == "agent-based":
        neighbor = self.generate_agent_based_neighbor(neighbor_size)
    elif select_command == "map-based":
        neighbor = self.generate_map_based_neighbor(neighbor_size)
    elif select_command == "uniform-random":
        neighbor = self.generate_uniform_random_neighbor(neighbor_size)
    else:
        raise NotImplementedError
```

```
cdef int[:] p = self.np_rng.permutation(neighbor.size()).astype(np.intc)
cdef ivec ans
ans.resize(neighbor.size())
cdef size_t i
for i in range(neighbor.size()):
    ans[i] = neighbor[p[i]]
return ans
```

如何进行单路径规划

- 可以参考根目录下的“单路径规划.pdf”有进一步说明
- 核心思路如下
  - 首先对于每个点（也可能是边），将其上面的时间分成若干段[0,t1), [t1,t2), … [tn, MAX_T)，表示交替的“连续被占用”和“连续无占用（安全段）”
    - 例如点1和点2有冲突，点2上有一段[2,3)的AGV路径片段，那么点1的分段就是[0,2), [2,3) [3,T)
  - 然后使用类似于最短路的方法，对上面的每一个时间段都找到无碰撞的最早到达时间（例如，点3的[5,10)时间段，最小的无碰撞到达时间为t=6），以及有一次碰撞的最造到达时间（这样如果无法找到无碰撞的到达终点的路径，难么这些依次碰撞的结果就会被用于下一次使用）
    - 例如，点10的安全段[3,10)最早在t=5.1到达，点10到点20的长度为4.5的边的安全段为[9,10)（此处边的安全段是对边的到达时间定义的），那么可以用t=5.1+4.5=9.6来更新点20的安全端[0,20)的无碰撞最早达到时间
    - 使用A*算法来减少拓展的节点数量增加效率
  - 最后对于终点的那些安全段找到最早到达时间