无人施工场景下的小车任务包调度框架及 ESP32车载终端实现

关键词：无人施工；多智能体调度；任务包；可视化仿真；ESP32

任务目标

- **在一个无人施工工地中，有多**辆小型施工小车负责物资搬运、材料配送和施工点补给。现场**没有人工**驾驶和人工调度，所有任务需要由后台系统自动分配和协调。

- **需要解决的**问题是：
  - 场景：无人施工工地，多辆小车负责物资搬运与施工补给。
  - 问题：没有人工调度时，如何自动为多辆小车持续分配和调整任务，使整体施工高效且负载均衡。
  - 约束：考虑施工配方、作业地点限制和车辆数量有限，同时预留接口接入真实 ESP32 小车进行联调。

解决方案

- 静态数据：用 SQLite 存施工区域、物资、配方、建筑需求

- C++ 仿真内核：
  - 维护世界状态 + 推进时间 + 执行任务：　　Simulator -> WorldState
  - 数据存储：　由python生成sqlite　->　用c++构建对象存储（对象类型信息在object.hpp）
  - 任务分配：　由分层图实现的　TaskTree
  - 调度模块：　以贪心的方式分配任务到各个小车的任务包（bundle）
    任务重或任务差的小车会尝试将任务交易给其他小车

- 视觉化检查：　调试和世界信息输出至 log 文件，　用 Pygame 做 2D 图形化检查

- ESP32 小车：通过 Wi-Fi 接收状态 / 指令，　用亮灯反馈任务类型

对任务的建模

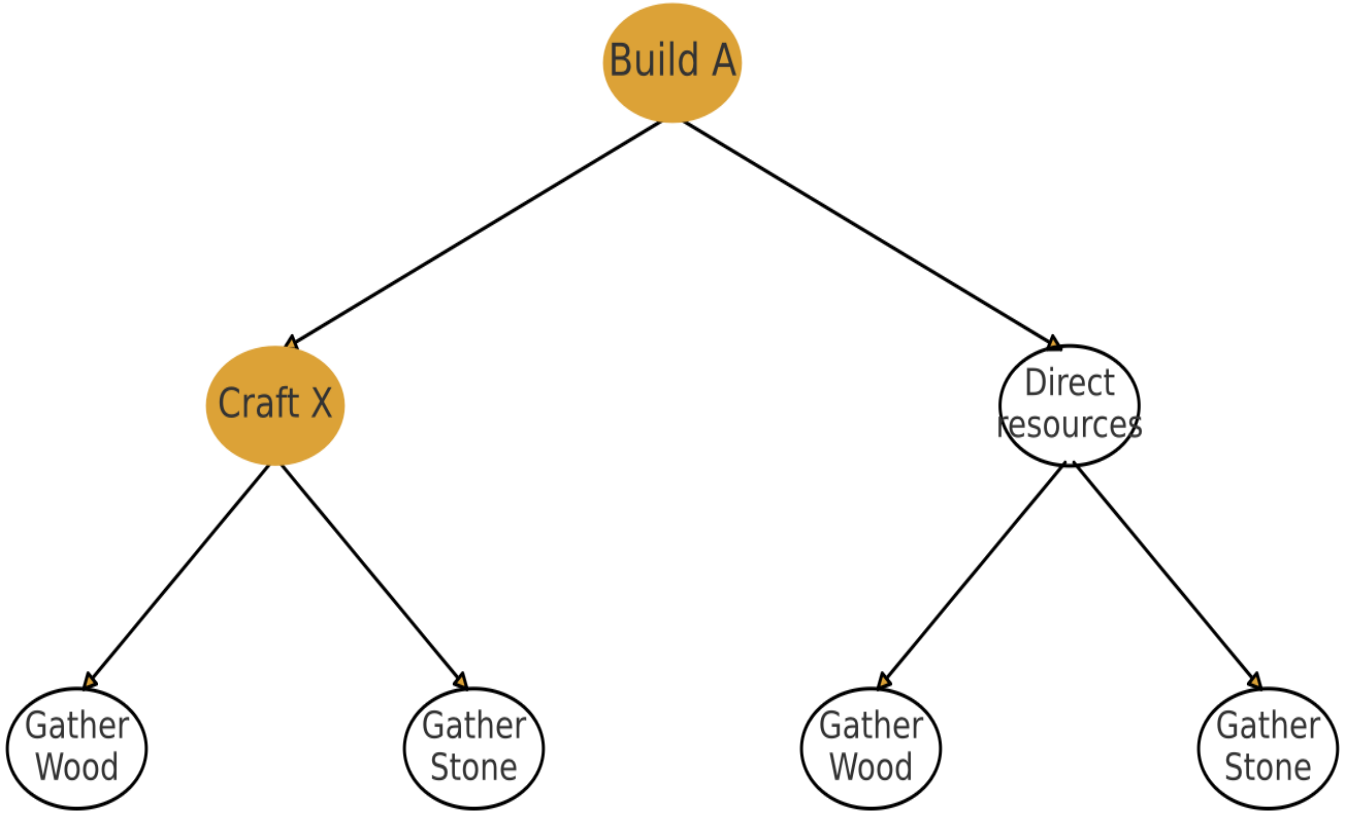
- 由于建筑任务是最复杂的，所以这里以建筑任务作为展示
- 考虑一栋建筑可能需要多种建材
- 建材可能需要材料制作
  - 建材的制作可能需要特定条件/地点
    - 举例：铁桶需要在铁匠铺制作，木板需要在木工坊制作。

- 我们可以将物品和建筑抽象为一个点，用箭头表示需求关系，则一个建筑任务的需求树如下一页图。

对任务的建模

- 不难注意到：
  - 底层的节点其实是相似的
  - 我们可以把相似任务合并

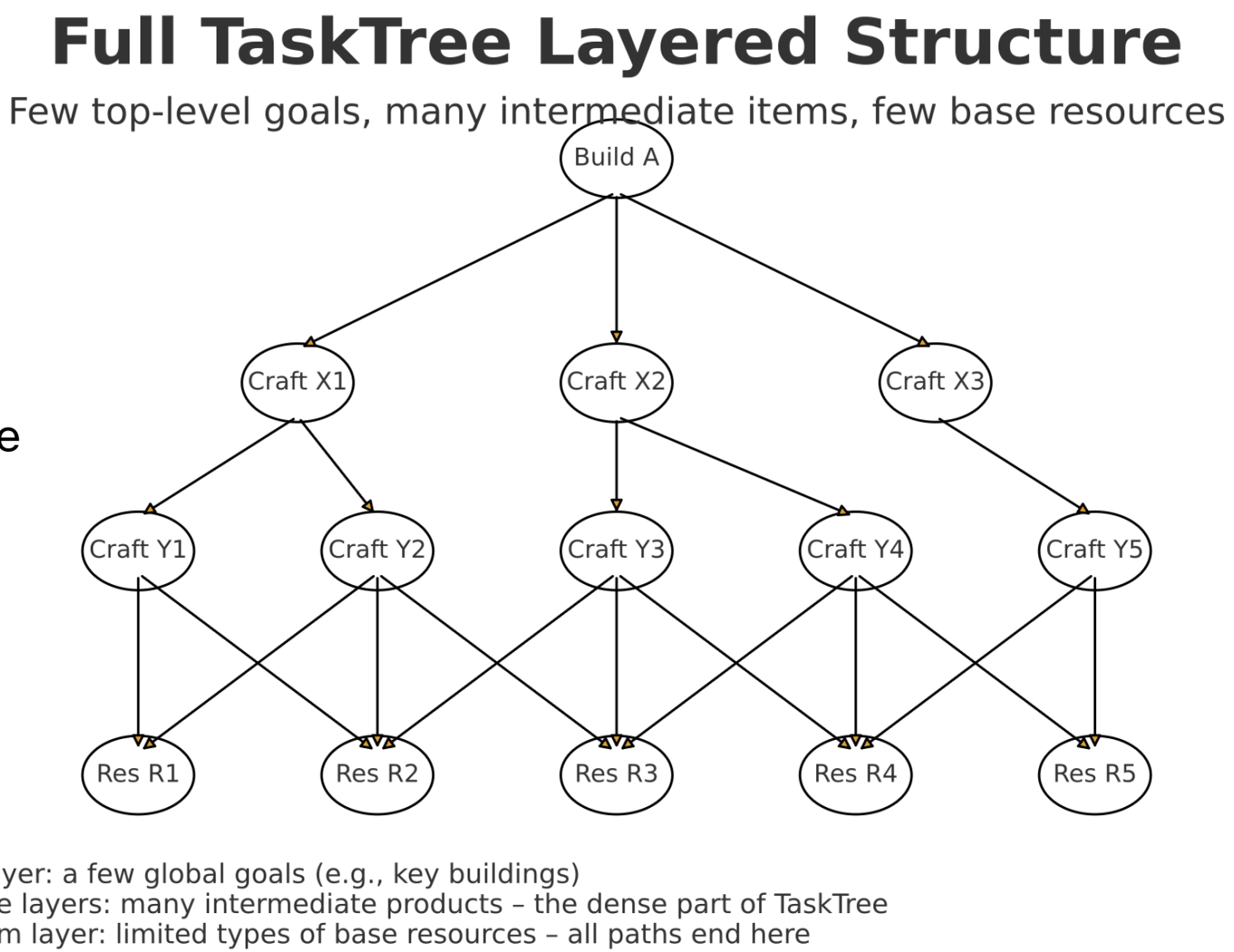
- 举个例子：
  - 1、需求3个木头
  - 2、需求5个木头
  - 可以合并为需求8个木头



- 当然，建筑任务，由于建筑需要在目标位置，所以不能合并
- 而采集任务比较特殊，也需要特殊处理。

对任务的建模

- 最终，我们会得到如右图的分层任务网络图（可以分层的有向无环图(DAG)）
- 其中当然，建筑不止一个，实际的TaskTree中，顶层会有更多点。

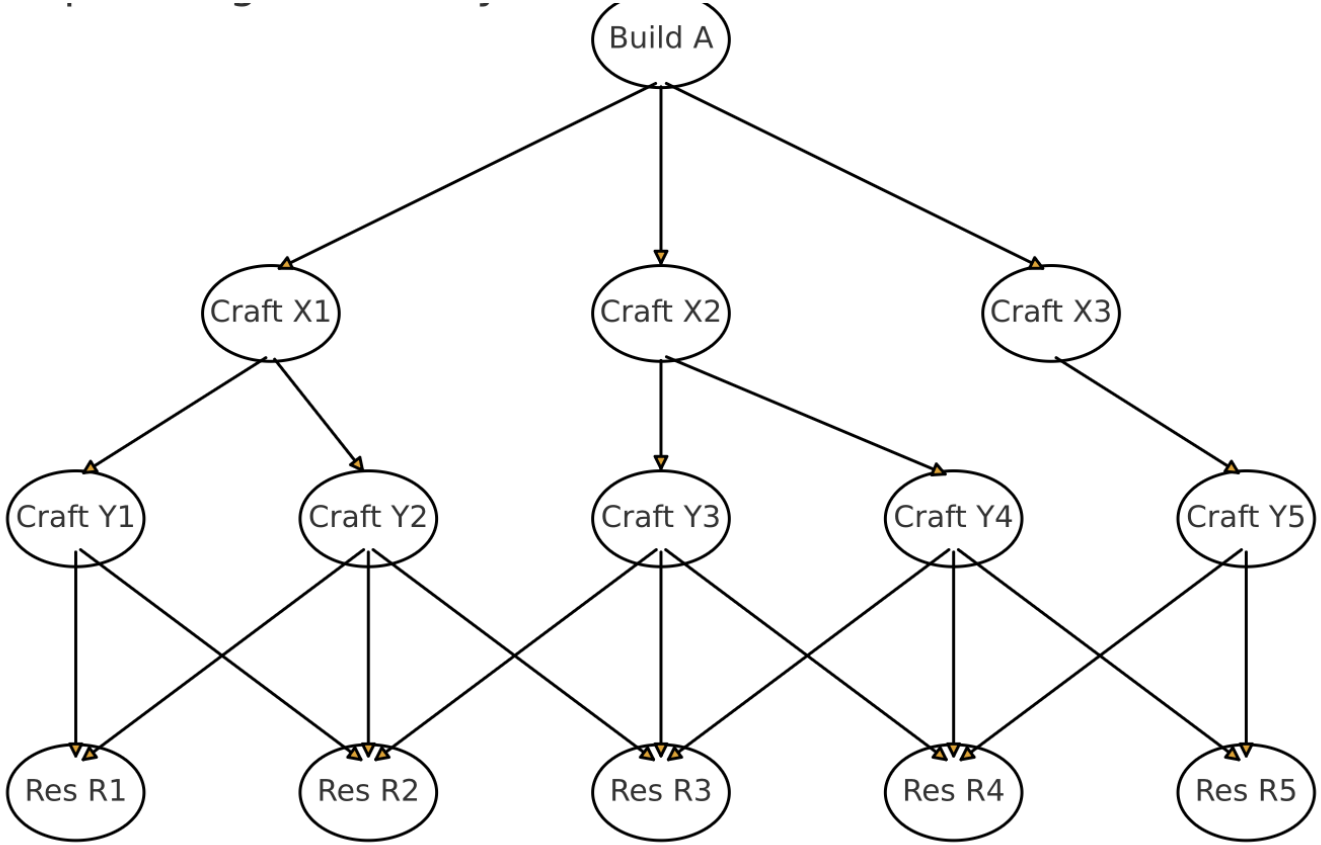


任务的生成

- **我**们会在树结构上进行一次搜索。

- **当（需求量** > 现有物资+预计产量**）**时
  说明这个物资有需求缺口
- **如果**这个物品可以生产。
- **此**时会生成一个生产任务。

- **搜索的**过程我在右图演示。



- **当然，被**约束条件限制的任务不会进行分配，会在下一次搜索时继续尝试。
- 约束条件指前置条件，材料不够并不会约束任务的生成，会在搜索时生成对材料的任务请求

任务的分配

- 当一个任务被生成时，会由小车们进行“竞标”
  - 简单来说，每辆小车预估一下自己拿到这个任务的得分
  - 得分最高的小车赢得该任务
  - 任务会存储在这个小车的bundle里面，并不会立即执行
  - 每辆小车可以存储多个任务。

- 当所有新任务都被分配后，小车会对自己的bundle进行一次排序，得出当前最优秀的选择
- 对于 「 预计完成时间 」 过长的小车，会尝试将自己低价值的任务交易给其他小车

分配的结果

- **尽可能能降低**时间浪费（move/standby）

- **最**终达到一个均衡负载的状态，尽可能快的完成建筑任务。

- **在有限的**时间复杂度内，得到一个任务调度的近似解。

  - （对于摆动情况，记录任务的交易次数，交易次数过多时会在近期小车中随机一个）

  - （**目前而言，随机退火**暂未加入，在此次演讲中，未标注为实现的功能均已在项目中实现）

估价的方式

- **任**务类型优先级：不同类型的任务有一个基础优先级
- **缺口**驱动：**当一个物品的缺口**过大时（或需求他的上级任务过多时）会有额外加分
- **距离**惩罚：对于需要去特定地点的任务，会有一个基于预计浪费时间的扣分
- **批次数和**负载均衡：
  - 对于需要往返的任务，**剩余往返次数越多**获得的加分越大
  - 对于Bundle过大**的小**车，会减去候选任务数量***固定数**值的分数

包内排序方式

- 目前的排序方式按照估价简单进行，在项目计划中，将在后续加入一个比较显然的贪心：
  - 目前存储bundle的方式是vector
  - 对于以下情况：

  - 简单预估A -> B的时间浪费和B -> A的时间浪费
  - 显然，C的内部对比较结果无影响， D的内部对比较结果无影响， E的内部对比较结果无影响，
  - 考虑C -> A， A -> D， D -> B， B -> E的时间开销
  - 优先时间开销小的

  - 当然，这个结果不严格正确，因为C，D，E与AB接触的地方可能会变动。

交易的过程

- 在每次任务分配后，会固定发起一次交易
- 对于「 完成bundle内任务 」 预估时间过多的小车，一定会尝试把自己所有的低价值任务交递出去。
- 当然，有些任务的价值就是很低，所以不能100%的保证全部交递。
- 对于平均水平的小车，会尝试将最差的几个任务交递出去。
- 随机抽取任务进行尝试。

手动分配优先级

- 对于任务的评分，**最后可以乘一个系数，或是直接覆盖**为inf
- 对于无关紧要的任务，可以*0.5或*0.1
- 对于优先的任务，**得分***10
- 对于置顶的任务，**得分覆盖**为最大值（目前为1e6）
- 对于当前材料不够的任务，得分*0，**无法制作**。

定期重分发

- 考虑到会出现某个任务卡住的情况，**会以固定**间隔对任务分配进行整体重制
- **重分**发时，所有的采集任务会被中断，正在进行的其它任务不会被中断
- 处于等待状态的任务会被回收，**并重新分配**

调试

- 本项目提供了非常多log输出（因为我在一个地方卡住了4小时，几乎把能输出的都输出了）
- 在输出部分有一个flag
  - flag=0表示不输出任何信息
  - flag=1表示输出世界信息
  - flag=2表示输出世界信息和调试信息

- 使用图形化展示功能要求输出世界信息，输出文件在项目根目录的Simulation.log

- 对于参数的修改，在README.md中可以找到详细说明

图形化展示

- 本项目用pygame做了个简单的图形化，允许用户以这种方式检查分配是否有问题。
- 其中：
  - 三角形表示资源点
  - 圆形表示NPC
  - 方框表示未完成的建筑
  - 实心方块表示已完成的建筑

- 所有实体均有描边，且颜色互不相同。

- 这个python程序由ai生成，所以我不做过多介绍。

ESP32 车载终端总体设计

- **定位**：作为“虚拟小车”的物理映射节点，用来验证调度结果能被真实硬件订阅和执行。
- **核心硬件**：ESP32 开发板 + 板载/外挂 RGB LED（可扩展接电机、传感器）。
- **运行环境**：ESP32 端运行轻量脚本（MicroPython / Arduino），上电自动连接指定 Wi-Fi

- 职责划分：
  - **上报**：周期性发送自身状态（如心跳、简单位置信息）。
  - **接收**：监听上位机发来的“任务状态/模式”指令。
  - 显示：用 LED 颜色/闪烁方式反馈当前任务类型。

- **在整体框架中的角色**：和 PC 端仿真、Pygame 可视化一起，构成“调度框架 → 图形化 → 实物终端”的完整链条。

Wi-Fi 通信与任务状态指示

- 通信流程：
  - ESP32 连接到与 PC 相同的 Wi-Fi 热点，获取本机 IP。
  - PC 端在可视化/仿真模块中配置 ESP32 的 IP 和端口，通过 UDP/TCP 发送状态字符串。
- 消息内容：
  - 由仿真/日志中选定一辆小车（如 NPC0）的“当前任务类型”。
  - 统一编码成简单文本："idle" / "gather" / "craft" / "build" 等。
- ESP32 端处理逻辑：
  - 监听指定端口，接收上位机发来的状态字符串。
  - 根据不同状态切换 LED 颜色，例如：
    - 空闲（idle）：灯灭或弱光；
    - 采集（gather）：绿色；
    - 制造（craft）：橙色；
    - 建造（build）：蓝色。

说明与演示

- 本项目是由演讲者**从零开始制作的，从构思到具体解决方案均是本人**设计，代码实现部分有用到ai辅助，但以演讲者为主导。

- **其中，C++代**码由演讲者主导，框架的设计，实现的方式均为人工设计，ai负责一些细节实现和code review
  - **有效代**码约1500+行。

- python代码完全由ai生成，python部分与核心代码无关，主要负责visualize和与ESP连接。
  - **其中，ESP32内的**boot.py约300行。
  - Visualization约300行。

- 本项目已的完整代码托管于 ：
  - https://github.com/lucaszou2000/traffic-simulation-coursework-2025