第3章

基于迭代局部搜索的启发式算法

过去的研究已经证明了敏捷卫星调度问题是一个 NP-hard 问题，因此精确算法难以快速求解大规模算例。在实际工程中，往往只要求算法的高效性和易实现性，而对解的质量没有过多的要求。这种情况下，启发式算法是一个很好的选择。针对时间依赖转换时间型和时间依赖收益型两种调度问题，我们均采用了迭代局部搜索（ILS）算法这一经典的启发式算法框架。ILS 算法是最经典也是最简单的迭代启发式算法，通常仅包含局部搜索算子和扰动算子。算法设计 ILS 算法，可根据特定问题特点制定专门的启发式算子，能有效解决许多复杂的组合优化问题，同时保留其参数少、效率高、易实现、通用性强等特点。

3.1 求解时间依赖转换时间型调度问题

针对时间依赖转换时间型敏捷卫星调度问题，本章提出了一种基于贪婪随机迭代局部搜索（GRILS）的启发式求解算法，该算法是在第 2 章中对转换时间的时间依赖特性建模分析的基础上进行的设计。算法的核心是能快速检查解可行性的插入算子，该算子能有效解决序列调度启发式搜索过程中容易出现的违反可见时间窗约束的情况，算子能准确并快速地判断在任务序列中任意位置插入新任务的可行性，并通过允许序列中已调度任务观测时间前移或推迟，使更多的任务可以被调度。

3.1.1 算法基本框架

贪婪随机迭代局部搜索启发式求解框架是贪婪随机自适应搜索步骤（greedy randomized adaptive search procedure，GRASP）和迭代局部搜索（ILS）的混合

算法[63]。其中，ILS 算法是最为经典的元启发式算法之一，已经被成功应用于很多组合优化问题中，具有通用性强、易实现等特点。GRASP 算法也是一种迭代算法，在每次迭代步骤中，在可调参数的基础上重新进行解的构造，并采用局部搜索对该解进行优化。这两种算法的混合版本，也就是 GRILS 算法，已经被用于高效求解带多时间窗的多约束团队定向问题（multi-constraint team orienteering problem with multiple time windows，MC-TOP-MTW）。而 MC-TOP-MTW 与敏捷卫星调度问题具有很多相似性，比如多时间窗约束、选择与调度决策等，本书采用了该启发式算法框架，但是针对我们的问题特点做出了很多改进。

具体来说，一般的 GRILS 算法可分为两层循环：外层循环是设置了“贪婪因子”这一可调参数的值，该参数控制了算法搜索的随机性，用于指导内层循环的算法求解；内层循环，实质上是一个迭代局部搜索的过程，在其一次迭代步内，基于外层设置的参数值，采用 GRASP 算法构建并优化一个解，如果得到的解更优则记录下来，随后采用扰动算子使搜索跳出局部最优。如果在一定连续步数内当前最好解得不到更新，则退出内层 ILS 循环，重新回到外层循环，设置新的参数。而在我们的 GRILS 算法中，为减少计算时间，每次 ILS 循环不会重新构造新的解，而是基于当前最好解，根据新的贪婪因子参数值进行迭代局部搜索。算法的核心是内层的 ILS，包含插入算子和扰动算子，外层循环只是用于调节指引 ILS 搜索方向，使搜索更多样化。

GRILS 算法的基本框架伪代码如算法 3.1所示，流程图如图 3.1所示。在外层循环中，算法控制贪婪因子参数 Greed(Greed $\in (0,1)$) 的值，该参数控制了内层 ILS 算法的随机性。Greed 越大，ILS 的随机性越强。内层的 ILS 由两个算子组成：插入算子 (INSERT( )) 和扰动算子 (SHAKE( ))。每一个 ILS 迭代步内，插入算子依次插入未调度的任务到当前解中，直至无法插入任何未调度目标为止。其中，选择哪些任务插入当前解中，会受到 Greed 参数的影响。扰动算子从当前解中移除一部分的已调度任务，避免搜索陷于局部最优。执行完插入算子后，如果新产生的解比当前最好解更好，则将其记录为新的当前最好解。如果在连续迭代步数 NumofIterNoImp 内，当前最好解都得不到提升，则结束内层 ILS 循环，回到外层循环。外层循环控制随机因子 Greed 从 StartGreed 开始，以 GreedDecrease 作为步长，逐步递减至 (StartGreed—GreedRange)。

算法 3.1 GRILS 算法框架

输入：观测目标集合 T

输出：最佳调度方案 S_b

$S_b \leftarrow \varnothing$;//当前最好解

$S_c \leftarrow \varnothing$;//当前解
Iteration $\leftarrow$ 0;
for Greed=StartGreed; Greed>StartGreed−GreedyRange; Greed=Greed−Greed-Decrease **do**
 $S_c \leftarrow S_b$;
 while Iteration < NumofIterNoImp **do**
  $S_c \leftarrow$ INSERT(S_c, Greed);
  if S_c 比 S_b 更优 **then**
   $S_b \leftarrow S_c$;
   Iteration $\leftarrow$ 0;
  else
   Iteration $\leftarrow$ Iteration+1;
  end if
  $S_c \leftarrow$ SHAKE(S_c);
 end while
end for
返回 S_b;

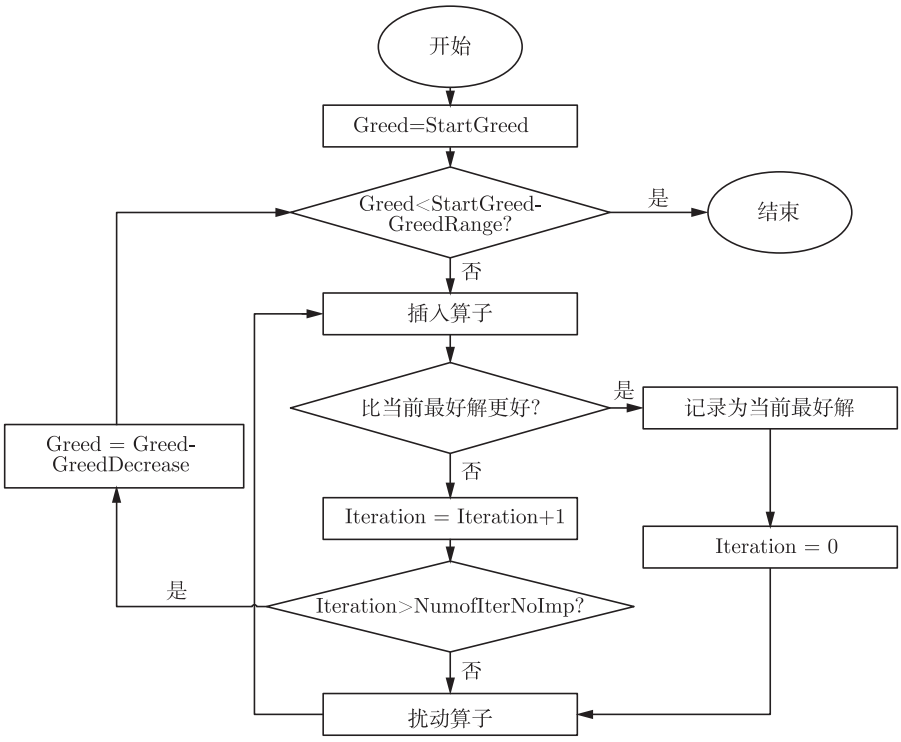

图 3.1　GRILS 算法流程图

3.1.2 插入算子

插入算子是本算法中最为核心的算子，直接用于提高调度总收益，优化当前解。每次调到该算子，算法从未调度目标中选择一个或多个插入当前解中，同时保证每次插入后解的可行性。敏捷卫星调度问题的解是由多个圈次的调度任务序列组成的。若给定某个任务序列，每一对连续观测任务之间必须满足转换时间约束和可见时间窗约束。为方便叙述，统一定义“插入”为将某指定目标插入某圈次任务序列的某指定位置的动作，可由三元组 $(i, \mathrm{Seq}, \mathrm{pos})$ 表示，其中 i 为待插入目标，Seq 为插入的任务序列，pos 表示插入位置。

假设某任务序列中每个任务的观测开始时间已确定，当往该序列的某个位置插入某个新任务时，若无法插入（违反转换时间和可见时间窗约束），则可以通过前移其前置任务或推迟其后继任务的观测开始时间，从而使该插入可行。这种允许序列中已调度任务前移或推迟的插入方式能有效提高搜索的有效性，然而也带来了一个问题：若插入位置相邻的两个任务的观测开始时间改变，与这两个任务相邻的其他任务有可能也需要重新调度，以确保解的可行性。也就是说，算法需要重新检查序列中其他对连续任务之间是否满足约束，并且重新确定其观测开始时间。若其中存在一对连续任务无法通过前移或推迟的调整方式保证解的可行性，则认为该插入不可行。这种“全”可行性检查的方式会造成计算时间的巨大浪费，这是因为算法搜索的过程中会存在大量不可行的插入尝试操作。如何使插入操作具备快速可行性检查能力，减少不必要的插入尝试次数，是插入算子设计的关键问题。

在最近的敏捷卫星调度研究中，Liu 等[22] 提出的自适应大邻域搜索（ALNS）算法采用了一种基于时间松弛量（time slack）的快速插入方法，该方法最早用于求解带时间窗约束的路径优化问题（VRPTW）[64]。在该方法中，序列中每个任务的前向（后向）时间松弛量表示该任务推迟 (提前) 的最大时间量，而不会影响其直接后继（前驱）任务的执行。通过推迟或前移插入位置相邻的任务，一个新任务可以很容易被插入，并且不会影响解的可行性，而时间松弛量界定了插入位置相邻任务的最大可横移时间量，可看作“局部松弛量”。然而，该方法忽略了很重要的一点，除了插入位置的相邻任务外，序列中其他任务也可以进行时间横移，从而容许更多的未调度任务插入序列中。这种基于序列“全局松弛量”的插入方法已经被应用于 OPTW 的启发式求解算法中[61]。在 OPTW 中，为实现可行性的快速检查，已调度节点序列中的每个节点都计算了一个时间松弛量 Maxshift，该松弛量代表了该节点执行时间能被推迟而不会使序列中后续所有节点违反约束的最大时间量。鉴于敏捷卫星调度问题与 OPTW 的相似性，本算法采用了该插入方法。

但不同之处有：① 本算法不需要确定任务序列中每个任务的具体观测开始时间，而只需确定每个任务的可行观测开始时间范围；② 转换时间的时间依赖特性需要考虑到该快速可行性检查中；③ 插入算子中定义了一个分配步骤 Assignment( ) 用于将重复调度到多个圈次序列的观测目标分配至其中一个圈次上。

图 3.2 的例子展示了插入算子如何在插入新任务时进行快速可行性检查。给定某个调度任务序列 $\mathrm{Seq}=\{(j-1),j,(j+1),(j+2)\}$，对其中每个任务（不妨假设为 j），从前往后计算其最早开始时间 $\mathrm{es}_j=\mathrm{EarliestStartTime}_{(j-1)j}(\mathrm{es}_{(j-1)})$，从后往前其最晚开始时间 $\mathrm{ls}_j=\mathrm{LatestStartTime}_{j(j+1)}(\mathrm{ls}_{(j+1)})$。序列中第一个任务 $(j-1)$ 的最早开始时间设为其可见时间窗的开始时间 $\mathrm{st}_{(j-1)}$，最后一个任务 $(j+2)$ 的最晚开始时间设为其可见时间窗的最晚可行开始时间，即 $(\mathrm{et}_{(j+2)}-d_{(j+2)})$。序列中的任务不需要确定其观测开始时间，因为 $[\mathrm{es}_j,\mathrm{ls}_j]$ 内任一时刻点都是任务 j 的可行观测开始时间。选定序列中任一任务的观测开始时间，可根据简单的回溯法确定其他任务的观测开始时间。

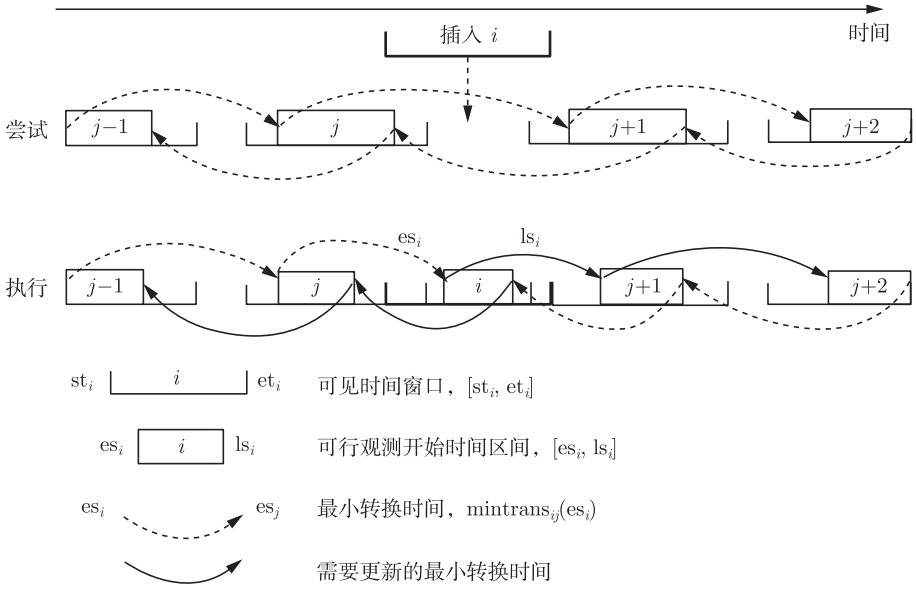

图 3.2　插入算子示意图

当往该序列的任务 j 和 $(j+1)$ 之间插入新任务 i 时，根据 $\mathrm{EarliestStartTime}_{ji}(\mathrm{es}_j)$ 计算其最早开始时间 es_i，根据 $\mathrm{LatestStartTime}_{i(j+1)}(\mathrm{ls}_{(j+1)})$ 计算其最晚开始时间 ls_i。显然，区间 $[\mathrm{es}_i,\mathrm{ls}_i]$ 定义了任务 i 在该插入位置的观测开始时间取值范围。只有当任务 i 的最早开始时间不晚于其最晚开始时间，即 $\mathrm{es}_i\leqslant\mathrm{ls}_i$

时，任务 i 插入该位置后不会使该序列不可行。若 $\mathrm{es}_i > \mathrm{ls}_i$，无论如何横移序列中的任务，也无法使该插入可行。这样的插入方式，一方面，避免了通过检查序列中其余任务的约束判断该插入的可行性，而只需检查待插入任务与插入位置相邻任务之间的约束，极大地减少了约束检查的次数；另一方面，由于时间依赖转换时间的存在，对任一对连续任务之间的约束检查都要求重新计算其最小转换时间，因此减少约束检查次数能极大地节省计算时间。另外，算法还可通过预处理阶段计算得到的“不可达”最早时间 ue_j 和最晚时间 $\mathrm{ul}_{(j+1)}$ 提前淘汰掉一部分插入尝试。以图 3.2 为例，若 $\mathrm{es}_j \geqslant \mathrm{ue}_j(i)$ 或 $\mathrm{ls}_{(j+1)} \leqslant \mathrm{ul}_{(j+1)}(i)$，则不需要计算任务 i 的最早开始时间和最晚开始时间，该插入位置必然不可行。

需要注意的是，这种插入方式只适用于转换时间满足 FIFO 规则和三角不等式规则的情况。若这两种规则不满足，则无法保证 $[\mathrm{es}, \mathrm{ls}]$ 区间内任一时刻都是可行观测开始时间。每次插入新任务之后，算法需要重新计算序列中插入位置之后所有任务的最早开始时间，以及插入位置之前所有任务的最晚开始时间。

插入算子 INSERT( ) 在尝试插入新任务时，先不考虑约束 (2.4)，即先允许每个目标可以在多个圈次上调度。在执行完插入操作后，当前解中可能存在某个目标对应多个任务，也就是说，该目标出现在多个圈次上的任务序列中。对这些“冲突”的任务，算法定义了一个 Assignment( ) 分配步骤，它可以检测出当前解中所有的“冲突”任务，并根据指定的启发式策略，对每个观测目标，当前解只保留其中一个任务，删除其余任务，并将删除任务对应的可见时间窗“冻结”。被标记“冻结”的时间窗不允许其参与到后续的插入操作中，除非通过扰动算子将其“解冻”。根据一些初步对比实验的结果，算法采用了一种窗口可用性分配策略，往往能取得较好的优化效果。该策略优先将重复调度的观测目标分配给拥有可见时间窗数量更少的圈次。通过这样的分配策略，拥有可见时间窗数量更多的圈次，就有足够的时间资源去容纳更多的未调度任务。采用 Assignment( ) 分配步骤的基本思想在于，不同圈次对任务序列的构造应当相对独立，如果某目标已经被调度在某圈次上，但在另一圈次的序列中又需要插入该目标，算法应当根据不同圈次的资源量 (如可见时间窗数)、已调度任务数等信息选择合适的圈次重新分配。

值得一提的是，基于 Liu 等[22] 的工作提出的求解多星调度问题的算法——基于自适应任务分配的 ALNS（adaptive task assigning based ALNS，A-ALNS）算法也包含了将观测目标按不同启发式规则分配至不同卫星的操作，分配后不同卫星调度的目标集合互不重合。实际上，若不区分卫星，而将所有卫星上的圈次资源

看作一颗卫星上的圈次，多星调度问题也能看作单星调度问题，因此本章提出的GRILS 算法也能求解多星算例。与 A-ALNS 算法的分配操作不同的是，GRILS 算法只关注于目标在不同圈次上的分配，而不是卫星层面的分配，分配粒度更细。此外，Assignemnt( ) 仅对当前解中会出现的被重复调度的目标进行分配，不需要对所有的目标预先分配，因此在算法的任何阶段，观测目标都可以被其任一可见圈次调度，而 A-ALNS 算法中，卫星分配后目标只能被调度到指定卫星的圈次上。

插入算子 INSERT( ) 的伪代码如算法 3.2 所示。首先，对任一圈次 k，对每一个未调度的或未“冻结”的可见时间窗 VTW_i^k，通过快速可行性检查的插入方法，对圈次 k 的任务序列 Seq^k 中每个位置，找出所有可行插入，并存储于列表 L_i^k 中。遍历完所有位置后，从 L_i^k 中选择代价最小的插入并置入圈次 k 的所有可行插入列表 L^k。这里的插入代价指的是在插入前后最小转换时间的变化量。以图 3.2 为例，插入任务 i 的代价为

$$\mathrm{cost}_i = \mathrm{mintrans}_{ij}(\mathrm{es}_j) + \mathrm{mintrans}_{i(j+1)}(\mathrm{es}_i) - \mathrm{mintrans}_{j(j+1)}(\mathrm{es}_j) \tag{3.1}$$

当圈次 k 中所有的未调度或未冻结窗口都遍历过后，从列表 L^k 按待插入任务的收益值从大到小排序，并保留前 $\lfloor(1-\mathrm{Greed})\cdot|L^k|\rfloor+1$ 个插入。从 L^k 中通过轮盘赌策略选出一个插入，并执行到当前解 S_c 的圈次 k 的任务序列中，更新序列中的最早开始时间和最晚开始时间。显然，参数 Greed 通过控制候选插入任务的选择范围来调整插入算子搜索的随机性，Greed 越小，收益小的插入更有可能被选择执行，随机性越强。所有圈次循环一遍后，通过 Assignment( ) 分配步骤，将当前解中被多次调度的目标分配到仅一个圈次上，从而保证当前解的可行性。

算法 3.2　INSERT(S_c, Greed)

输入：当前解 S_c 和贪婪因子 Greed
输出：当前解 S_c
while TRUE **do**
 for 任一圈次 $k \in O$ **do**
  for 对圈次 k 上的任一未调度或未冻结的 VTW_i^k **do**
   设 S_c 中圈次 k 的已调度序列为 Seq^k;
   for 对序列 Seq^k 中任一插入位置 pos **do**
    设 pos 在任务 j^k 和 $(j+1)^k$ 之间;
    if $\mathrm{ue}_j^k(i) > \mathrm{es}_j^k$ 且 $\mathrm{ul}_{(j+1)}^k(i) < \mathrm{ls}_{(j+1)}^k$ **then**
     $\mathrm{es}_i^k \leftarrow \mathrm{EarliestStartTime}_{ji}(\mathrm{es}_j^k)$;
     $\mathrm{ls}_i^k \leftarrow \mathrm{LatestStartTime}_{(j+1)i}(\mathrm{ls}_{j+1}^k)$;

```
            if es_i^k <= ls_i^k then
              将该插入置入列表 L_i^k 保存;
            end if
          end if
        end for
        从列表 L_i^k 选择代价最小的插入并置入圈次 k 的列表 L^k;
      end for
      将列表 L^k 按收益从大到小排序，并删除后 ⌈|L^k|⌉ − 1 个插入;
      根据收益值，通过轮盘赌策略从 L^k 中选择一个插入并在当前解 S_c 中执行;
      更新该插入位置之后 (之前) 的所有任务的最早开始时间 (最晚开始时间);
    end for
    执行 Assignment(S_c);
    if 对所有圈次 k, L^k = ∅ then
      返回 S_c, 退出算子;
    end if
  end while
```

3.1.3 扰动算子

设置扰动算子 SHAKE( ) 的目的在于避免解的搜索陷入局部最优，增加搜索方向的多样性，从而在后续的局部寻优中获得更高质量的解。该算子同样参考了 Vansteewegen 等[61] 关于求解 OPTW 的工作。每次调用该算子，当前解中每个圈次上的任务序列都会被移除一个任务子序列，即移除一个或多个连续任务。该算子的执行依赖于两个整数参数向量 $\boldsymbol{S}_d$ 和 $\boldsymbol{R}_d$。其中，$S_d(k)$ 表示圈次 k 的任务序列中被移除的子序列的起始位置，$R_d(k)$ 表示圈次 k 的任务序列中被移除的子序列的规模。

在内层 ILS 算法开始时，初始化 $\boldsymbol{S}_d$ 和 $\boldsymbol{R}_d$ 的所有元素为 1。当调用扰动算子时，对每个圈次 k，从其调度序列的位置 $S_d(k)$ 开始删除 $R_d(k)$ 个连续调度任务，若删除时超过末位的任务，则从第一个任务继续删。每次执行完扰动算子，令 $S_d(k) = S_d(k) + R_d(k)$，$R_d(k) = R_d(k) + 1$。如果 $S_d(k)$ 大于圈次 k 的已调度任务数，则令其减去任务数，使其回到序列靠前的位置。如果 $R_d(k)$ 大于任务数的 1/3，则令 $R_d(k) = 1$。若当前 ILS 迭代步内，当前解的质量比当前最好解要好，则令 $R_d(k) = 1$。这种动态调整参数 S_d 和 R_d 的任务移除方式，很大程度上使序列中每一个任务都至少被移除一次，使后续的迭代局部搜索更多样化，更全面地搜索整个解空间。这种扰动技巧已经被成功应用于一些经典问题的启发式求解算法中[65]。

3.2　求解时间依赖收益型调度问题

时间依赖收益型敏捷卫星调度问题是在时间依赖转换时间型调度问题基础上，考虑观测目标在其可见时间窗内不同时刻观测所获收益不同的情况。时间依赖收益特性源于敏捷卫星的观测成像质量与其观测姿态角度密切相关，而可见时间窗内不同时刻点对应不同的观测姿态角。针对上述问题特点，本节提出了一种双向动态规划-迭代局部搜索（BDP-ILS）的启发式求解算法。其中，迭代局部搜索（ILS）算法用于构造可行任务序列，与 GRILS 算法思想基本一致。算法的核心是双向动态规划方法，用于优化任务序列的观测开始时间，快速评估其最大实际收益，并为启发式搜索提供指导信息。

3.2.1　求解思路与算法框架

时间依赖收益型调度问题的优化目标与两个方面的因素相关：① 选择和调度哪些观测目标；② 调度任务的观测开始时间。对该问题的求解思路可以分两步走：首先，采用启发式算法选择和调度观测目标，构造多个圈次上的可行任务序列；其次，优化序列中每个任务的观测开始时间，以获得该序列的最大观测收益。这样的求解思路实质上是将复杂的原问题转化为两个相对简单的优化子问题。第一个优化子问题是从一组候选观测目标中选择一部分进行调度，生成任务序列集合，并满足转换时间约束、时间窗约束和目标唯一性约束等，此时可采用本章第一部分所提出的 GRILS 中的内层 ILS 来完成。第二个优化子问题则是，在保证任务序列不变和约束可行的情况下，确定每个任务的观测开始时间，并最大化实际收益，其实质是对任务序列的收益值评估。任务序列的实际收益是指在保证序列中任务及其顺序不变的情况下，通过确定每个任务的观测开始时间，得到整个序列的总收益。对该决策问题的求解是整个时间依赖收益型调度问题的关键，也是算法设计的核心。我们发现了该问题具有最优子结构，可以通过动态规划方法在多项式时间内求解，在下一小节中会详细介绍。

需要注意的是，这两个优化子问题是相互耦合相互影响的。第一个子问题构造出来的任务序列，是第二个子问题的输入，限定了序列中任务观测开始时间的取值范围，决定了第二个子问题的最优值，即最大实际收益；第一个子问题中，每次通过局部搜索算子改变任务序列时，需要使用第二个子问题的优化算法来评估改变前后任务序列的最大观测总收益的差异，才能判断该局部搜索算子对当前解的影响。

综合以上分析，本节提出了一种双向动态规划-迭代局部搜索 (BDP-ILS) 的启发式求解算法，算法框架伪代码如算法 3.3所示。BDP-ILS 算法的基本框架是迭代局部搜索 (ILS)，仅包含插入算子 INSERT-BDP( ) 和扰动算子 Shake( )。其中，扰动算子与时间依赖转换时间型调度（GRILS）算法的扰动算子一致，不同的是，扰动完成后需要采用一种双向动态规划方法来优化当前解中每个任务的观测开始时间，评估当前解的最大观测收益，该评估算子用 FullEvaluation( ) 表示。插入算子 INSERT-BDP( ) 是在 GRILS 算法的插入算子的基础上，通过上述相似的双向动态规划方法评估插入操作对当前解的最大观测收益的影响。但不同的是，算法通过记录序列中每个任务的前向（后向）累积收益函数，实现了对插入操作的快速评估，为了区分，该评估算子记为 FastEvaluation( )。若当前最好解连续 NumofIterNoImp 迭代步都无法得到提升，则算法终止，输出当前最好解。

算法 3.3 BDP-ILS 算法框架

输入：观测目标集合 T
输出：最佳调度方案 S_b
$S_c \leftarrow \varnothing$;//当前解
$S_b \leftarrow \varnothing$;//当前最好解
Iteration $\leftarrow 0$;
while Iteration $<$ NumofIterNoImp **do**
  $S_c \leftarrow$ INSERT-BDP(S_c);
  if S_c 比 S_b 更优 **then**
    $S_b \leftarrow S_c$;
    Iteration $\leftarrow 0$;
  else
    Iteration $\leftarrow$ Iteration $+ 1$;
  end if
  $S_c \leftarrow$ Shake(S_c);
  FullEvaluation(S_c);
end while
返回 S_b;

3.2.2 双向动态规划评估

双向动态规划算法求解的优化子问题是，给定某个任务序列，在不改变序列可行性的情况下，优化并确定每个任务的观测开始时间，使序列的总观测收益值最大化。该问题具有最优子结构，即原问题的最优解包含其子问题的最优解，而子问题具有和原问题类似的性质。此时，可以采用递归的方法自底向上地求得子