arXiv:2302.00776v1 [cs.AI] 1 Feb 2023

带运动学与动力学约束的安全区间路径规划

Zain Alabedeen Ali, [1] **Konstatin Yakovlev** [2, 3, 1]

[1] 莫斯科物理技术学院，俄罗斯莫斯科 [2] 俄罗斯科学院计算机科学与控制联邦研究中心，俄罗斯莫斯科[3] AIRI，俄罗斯莫斯科 ali.za@phystech.edu，yakovlev@isa.ru

摘要

安全区间路径规划（SIPP）是一种强大的算法，可用于解决单智能体寻路问题：智能体被限制在图中活动，而由于环境中的动态障碍物，图中的某些顶点/边会在特定时间区间内被阻塞。原始 SIPP 算法依赖于这样一个假设：智能体能够瞬时停止。然而，这一假设在实践中往往并不成立。例如，以巡航速度行驶的移动机器人无法立即停下，而是需要逐渐减速才能完全停止，这需要一定时间。换言之，机器人受到运动学与动力学约束的限制。遗憾的是，正如我们在本文中所展示的，在这种情况下，原始 SIPP 并不完备。为此，我们提出了一种新型 SIPP 变体，并证明其在考虑加速/减速的规划中是完备且最优的。实验评估表明，原始 SIPP 的关键优势在修改后的版本中依然成立——与 A*算法相比，它所需扩展的节点少得多，因此速度显著更快。

引言

在存在静态和移动障碍物的情况下规划路径，是一个颇具挑战性的问题，在机器人学、电子游戏及其他领域都有重要应用。当环境完全可观测，且已知动态障碍物的轨迹时（例如，由机器人的感知系统或全局观测器预测得出），在规划时将其纳入考虑是合理的。专为这类情形设计的一种典型方法是安全区间路径规划（SIPP）（Phillips 和 Likhachev，2011年）。这是一种基于搜索的算法，在图上运行：图的顶点对应智能体的构型（例如位置、朝向、速度等），边则对应这些构型之间的转换。由于动态障碍物的存在，该图中的某些顶点或边会在特定时间区间内被阻塞。SIPP 会将这一点纳入考虑，并构建允许智能体在顶点处等待以避免碰撞的规划。SIPP 的一个明确假设是，智能体可以启动或停止移动——

瞬间完成。在这一假设下，可以证明该算法是完备且最优的（相对于给定的空间和时间离散化）。

在我们的工作中，我们采用了一种（从实际角度来看更现实的）假设，即减速直至完全停止

这是被 AAAI 2023 接收的论文预印本。

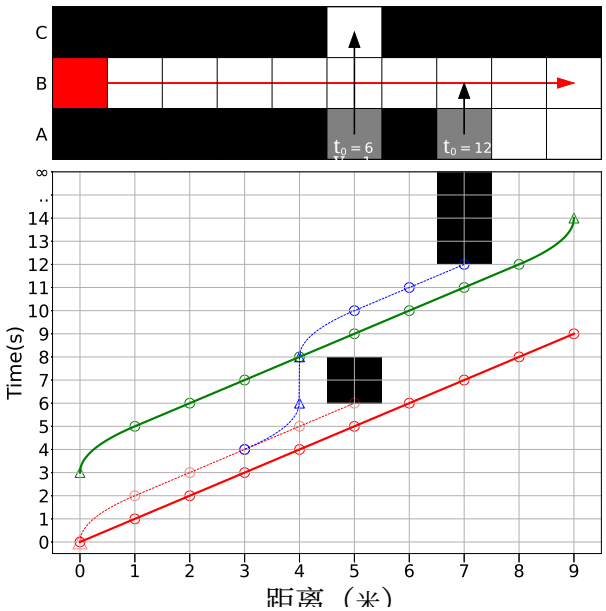

图 1：一个包含动态障碍物的复杂路径规划实例。

同样，从完全停止状态加速也需要时间。换言之，智能体受运动学与动力学约束。这使得 SIPP 无法适用，如图 1 所示。在此示例中，环境被离散化为 3×10网格，网格单元中心之间的距离为 1 m，智能体需要从最左侧的单元格 $B0$ 前往最右侧的单元格 $B9$。此外，环境中还存在两个动态障碍物，它们会在图下方黑色部分所示的时间段内阻挡单元格 $B5$ 和 $B7$。假设智能体的巡航速度为 1 m s^{-1}。以此速度在相邻单元格之间移动需要 1 s。此外，智能体的最大加速度/减速度为 0.5 m s^{-2}，这意味着从完全停止加速至巡航速度需要 2 s，行进距离为 1 m。因此，从完全停止状态开始移动时，智能体到达相邻单元格需要 2 s，而非 1 s。如果算法忽略这一点（就像假设加速度瞬时完成的原始 SIPP 一样），那么生成的规划（以红色显示）实际上会导致碰撞（如果在实际运动学与动力学约束下执行）。

如红色虚线所示，若考虑实际运动学与动力学约束，直接修改 SIPP 以纳入加速/减速动作的一种简单方法是仅采用在动力学上有效的转移，即不允许在速度不为零的状态下等待。然而，这一变体会报告无法找到解，同时生成如蓝色所示的部分规划。在此规划中，智能体成功在第一个障碍物前停下并等待了 2 s，但随后无法通过第二个障碍物，因为它到达 $B7$时已经太晚。然而，确实存在无碰撞规划，如绿色所示。SIPP 之所以无法找到该规划，直观来说，是因为 SIPP 会推迟等待动作，无法推理在搜索树的不同状态下等待所造成的后果。当任何状态下都可以等待时，这并不会违反理论保证。另一方面，如果并非总能随时等待，即智能体需要时间才能停下，那么该算法显然就不完备。我们将在论文后续章节对此作进一步阐述并提供技术细节。

据我们所知，尚无关于 SIPP 的研究直接解决这一问题。因此，在规划时必须考虑智能体的运动学与动力学约束的情况下，所有已知的基于 SIPP 的算法都不完备。我们的工作填补了这一空白，并提出了一种在此类约束下可证明完备且最优的算法——带区间投影的安全间隔路径规划（SIPP-IP）。为进行实证评估，我们开展了广泛的实验，将 SIPP-IP 与若干基线方法进行了比较，其中包括其他可能想到的（非完备）SIPP 变体以及 A*算法。实验结果清楚地表明，SIPP-IP 优于所有这些方法：它能够解决其他规划器无法解决的实例，且其运行时间比 A*算法低两个数量级。

相关工作

对于存在可预测移动障碍物的情形，基于搜索的规划可以直接通过带时间离散化的 A*算法实现（Hart、Nils- son 和 Raphael，1968年）。然而，考虑时间维度会导致搜索空间显著增大，尤其是在需要精细时间离散化时。为此，Phillips 和 Likhachev（2011年）提出了 SIPP，其核心思想是将一系列时间步压缩为时间间隔，并在这些间隔上进行搜索。在智能体能够瞬时开始或停止移动等若干假设下，SIPP 可证明是完备且最优的。此后，出现了许多 SIPP 变体，用以改进原始算法，例如任意角度SIPP（Yakovlev 和 Andreychuk，2021）、随时SIPP（Narayanan、Phillips 和 Likhachev，2012），以及多种有界次优 SIPP 变体（Yakovlev、Andreychuk 和 Stern，2020）。此外，SIPP 及其改进版本还被广泛用作一些最先进的多智能体寻路求解器的构建模块（Cohen 等人，2019；Li 等人，2022）。所有这些变体都没有考虑运动学与动力学约束。

接下来，我们只介绍那些在一定程度上考虑智能体运动学和/或运动学动力学约束的工作。在（Ma 等人，2019）中，研究者提出了 SIPPwRT，使规划能够支持不同速度。

不过，这项工作并未考虑加速动作及其影响。类似地，（Yakovlev、Andreychuk 和 Vorobyev，2019）介绍了 SIPP 的任意角度变体，该变体支持非均匀速度。有趣的是，作者在真实机器人上评估了他们的规划器。为了给这些机器人规划安全轨迹，他们提出扩大移动障碍物的尺寸；换个角度看，这相当于延长某些图顶点/边的阻塞时间区间。这种临时方法在实践中表现尚可，但一般而言，它引出了如何选择偏移量的问题。例如，在引言所述的场景中（回顾图 1），如果将阻塞区间延长 1 秒，SIPP 就无法找到解。在（Ali 和 Yakovlev，2021）中，加速动作被直接整合进 SIPP，但这同样会使算法不完备（本文将对此进行说明）。（Cohen 等人，2019）针对一种假设任意运动模式的问题设置提出了 SIPP 的一个变体，但这并非论文的重点。因此，作者没有提出专门处理加速运动的技术，实证评估也只考虑了起始和结束时速度均为零的运动。总而言之，据我们所知，在本研究之前，还不存在这样一种 SIPP 变体：当原始 SIPP 所假设的“机器人可以瞬时启动/停止”不成立时，它仍能被证明是完备且最优的。

问题陈述

我们假设存在一条离散时间线 $T = 0, 1, \ldots\ldots$。智能体关联着一个图 $G = (V, E)$，其起点和目标顶点为：$start, goal \in V$。图中的一个顶点对应于智能体的状态，也称为构型，在该状态下，智能体可以停留而不与静态障碍物发生碰撞。每个构型都明确包含智能体的速度 vel。例如，构型可以由智能体的坐标、朝向和速度组成：$v = (x, y, \theta, vel)$。位置/朝向相同但速度不同的构型，确实对应于 V 中不同的顶点。具有$vel = 0$ 的构型尤为重要，因为智能体只有在这些构型下才能原地停留（等待）。

一条边 $e = (v, u) \in E$ 表示一个运动基元（Pivtoraiko 和 Kelly，2011）——智能体运动中一段较短且符合运动学和动力学约束的片段，可将智能体从 v 转移至（不同的） u。我们假设，对于任意 $v \in V$，都有有限个此类运动可供选择，且每种运动所需的时间步数均为整数。在实际应用中，这一假设并不苛刻，因为任何实数持续时间都可以通过一定精度进行近似，并表示为整数。运动的持续时间决定了边的权重，$w(e) \in \mathbb{N}$。

根据源/目标速度，每个运动（边）都可以分为加速、减速或匀速。在后一种情况下，智能体在目标构型处的速度与源构型处相同；而在前两种情况下，速度会发生变化（分别增大和减小）。加速/减速运动的存在使我们的问题区别于那些假设加速度瞬时发生的研究，后者

假设加速度是瞬时的。[1]

除了智能体和静态障碍物外，环境中还有固定数量的移动障碍物。我们假设这些障碍物的轨迹已知，并通过辅助程序转换为与图顶点和边相关的碰撞区间和安全区间，即，对每个顶点，都给定一个由有限个互不重叠的时间区间组成的序列$SI(v)$，该序列表示智能体可以在 v 配置于此且不会与任何移动障碍物发生碰撞的安全区间。同样，也为每条边定义安全区间， $SI(e)$。如果智能体在任何不属于 $SI(e)$的时间步执行移动 e，就会与移动障碍物发生碰撞。

智能体的一条（定时）路径，或称轨迹，是一个序列 $\pi = (e_1, t_1), (e_2, t_2).....(e_L, t_L)$，其中 $t_i \in T$是由边 e_i 定义的运动开始的时刻。轨迹的代价是智能体到达最终顶点时的时间步， $cost(\pi) = e_L + w(e_L)$。如果 e_i 的目标顶点与e_{i+1} 的源顶点相匹配，且 $t_{i+1} \geq t_i + w(e_i)$（带有 $t_0 = 0$），则该轨迹是可行的。此外，如果 $t_{i+1} > t_i + w(e_i)$，那么 e_{i+1}源点处的速度必须为零。

可行轨迹也可以看作由移动动作和等待动作组成的规划，其中移动动作由图边定义，而等待动作可能发生在智能体速度为零的顶点处。第 i 个移动动作之后发生的等待动作的持续时间计算为 $\delta = t_{i+1} - (t_i + w(e_i))$。由于 $w(e_i)$ 是整数，因此 δ 也是整数。

轨迹 π 与动态障碍物发生碰撞当且仅当 以下任一条件成立：*i*) 智能体在不属于 $SI(e)$ 的时间步开始执行某个运动基元 e ； *ii*) 根据 π，智能体在顶点 v 等待，而该顶点不属于 $SI(v)$。

现在的问题是在给定的 G（其中标注了顶点/边的安全区间）上，寻找一条可行且无碰撞的轨迹 π，将智能体从 $start$ 转移到 $goal$。本文旨在最优地解决这一问题，即尽可能早地到达目标。

方法

由于我们的方法依赖于 SIPP，而 SIPP 又依赖于带时间步的 A*算法，因此我们先介绍背景，然后详细讲解所提出的方法。我们假设读者熟悉用于静态图的基础 A*算法。

背景

带时间步的 A*算法解决所考虑问题的一种直接方法是使用 A*算法搜索状态空间，其节点是元组 $n = (v, t)$，其中 v 是图顶点， t 是到达该顶点时的时间步。展开一个节点时，按如下方式生成后继节点。首先，对于每个 $e = (v, u)$，如果 e 可行，则将节点 $n_{move} = (u, t + w(e))$ 添加到后继节点中。事实上，如果在时间 t 执行运动 e 会导致碰撞，则丢弃生成的节点。其次，如果 v 允许等待，则还会将对应于持续时间最短的等待动作（一个时间步）的节点 $n_{wait} = (v, t + 1)$ 添加到后继节点中。同样，如果 v 的安全区间未覆盖时间步 $t + 1$，则丢弃该节点。

搜索算法的其他部分与 A*算法完全相同。值得注意的是，节点的 g值等于其标识符中的时间分量。也就是说，节点 $n = (v, t)$ 的 g值等于 t，因为这是从 $start$到 v的代价的最小已知估计（相对于算法当前迭代）。

SIPP 在许多场景中，带时间步的 A*算法会生成大量形如 $(v, t), (v, t + 1), (v, t + 2)$ 等的节点，这些节点是通过使用原子等待动作创建的。这会导致搜索树增大并减慢搜索速度。为此，SIPP 采用将连续的等待动作压缩的思路，以减少需要考虑的搜索节点数量。

SIPP 的搜索节点表示为 $n = (v, [lb_j, ub_j])$，其中 $[lb_j, ub_j]$ 是 v 的第 j 个安全区间。对于每个节点，SIPP 都会存储其最早到达时间： $t \in [lb_j, ub_j]$。每当找到一条通往 n 的代价更低的路径时，就会更新 $t(n)$ ； 与带时间步的 A* 算法类似，该值用作节点的 g 值。

扩展节点时，SIPP 不会生成与等待动作对应的独立后继节点。相反，它只会生成与“等待并移动”动作对应的后继节点，这些动作会使智能体到达邻近构型的安全区间内。也就是说，为了生成后继节点，SIPP 会等待尽可能短的时间，以便在移动到新构型时尽早到达新的安全区间。例如，如果 SIPP 执行一次从节点 $n = (v, [5, 10])$ 到节点 $n' = (v', [15, 18])$ 的移动，且该移动的持续时间为 5，并且 $t(n) = 7$ ，那么 $t(n')$ 就是 15。这意味着智能体在 $t = 7$ 到达 n 后，会在那里等待 3 个时间步，并于 $t = 10$ 开始移动，在 $t = 15$ 到达 n'。该移动不能早于 $t = 10$ 执行，否则将会在 n' 的安全区间之外结束。

SIPP 比 A*算法快得多，并且在以下假设下可证明是完备且最优的：忽略惯性效应，智能体可以瞬时开始和停止运动，因此所有构型都可以执行等待动作。

如果等待动作并非总是可用呢？ SIPP 内在假设智能体可以在任意构型下等待，但这一假设在实践中往往不成立，因为许多移动智能体无法瞬时开始或停止运动。为反映这一点，应引入加速和减速运动，并将速度变量加入智能体的构型中（如我们在问题陈述中所做的那样）。在这种设定下，SIPP 不完备。

陈述 1. 当智能体受到运动学与动力学约束，且在任何构型下都无法执行等待动作时，*SIPP* 并不完备。

[1]请注意，我们的问题表述和所提出的方法均不依赖于智能体执行运动基元时状态变量（例如速度）的变化方式。我们只关注运动基元端点处的状态变量取值。

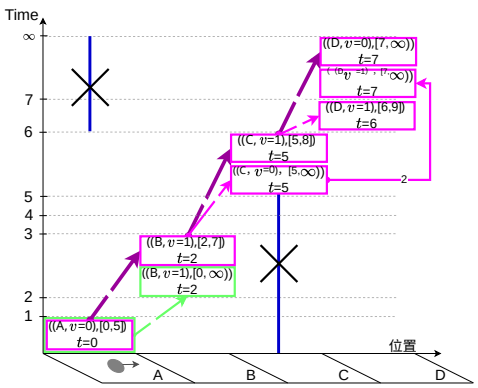

图2：一个标准 SIPP 无法找到解，而 SIPP-IP 成功找到解的示例。SIPP 的搜索树以绿色显示，SIPP-IP 的搜索树以紫色显示。

证明。 我们通过给出一个示例来证明该陈述：尽管存在解，SIPP 却无法找到它。考虑图 2 所示的问题。智能体需要从单元格 A 到达单元格 D。运动原语定义如下：

- (加速运动)智能体从一个单元格出发，以 $vel = 0$ 开始运动，并以 $vel = 1$ 结束于相邻单元格。代价（持续时间）为 2 个时间步。
- (匀速运动) 智能体从一个单元格出发，带着 $vel = 1$开始移动，并在相邻单元格中以 $vel = 1$结束。代价（持续时间）为 1 个时间步。
- （减速运动）智能体从一个单元格出发，以 $vel = 1$ 开始移动，并以 $vel = 0$ 结束移动，抵达相邻单元格。代价（持续时间）为 2 个时间步。

实际上，当 $vel = 0$ 时，智能体可以等待。等待动作的持续时间为 1 个时间步。为简洁起见，我们忽略智能体的朝向以及会改变其朝向的运动。

SIPP 首先扩展初始搜索节点 $((A, vel = 0), [0, 5])$，其中只有一个运动基元适用，即加速度运动。这会生成后继节点 $((B, vel = 1)\ [0, \infty))$，其到达时间为 $t = 2$，如图 2 中的绿色部分所示。智能体无法在此构型下等待（因为其速度不为 0）。因此，它只能继续移动，使用匀速运动动作在 $t = 3$ 时到达 C，或使用减速运动在 $t = 4$ 时到达。两种移动都无效，因为它们到达 C 时不在其安全区间内，该区间从时间 5 开始。因此不会生成后继节点，SIPP 搜索树中也没有可供扩展的搜索节点。算法随即终止，未能找到解。然而，实际上存在解：智能体需要在 A 等待 2 个时间步，然后继续朝 D 移动。这样，它将在 $t = 5$ 到达 C，满足安全区间约束。 □

接下来，我们将介绍一种 SIPP 的改进方法，在所考虑的假设下，该方法是完备且最优的。

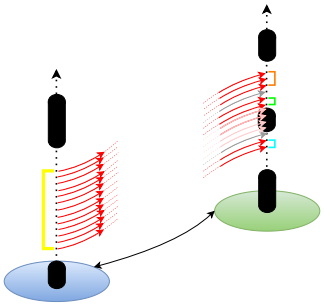

图3：SIPP-IP 中时间区间的投影。

SIPP-IP：带（等待）区间投影的安全区间路径规划

思路 标准 SIPP 无法解决上述规划实例这类问题，主要是因为有关可能等待动作的信息不会从前驱节点传递到后继节点。在所考虑的示例中，当到达搜索节点 $((B,\ vel = 1)\ [0, \infty))$ 时，SIPP 会“忘记”智能体可以在前驱节点处等待，并在前驱节点的安全区间结束之前的任意时间步执行移动动作。如果智能体可以在任意构型处等待，这就不是问题；但在我们考虑的情形下，这会导致算法不完备。

为此，我们用另一个时间区间替代安全区间，作为搜索节点的标识符，我们将其称为等待区间。对于任意搜索节点，该区间都属于其图顶点的安全区间。实际上，具有相同图顶点但等待区间不同的节点应当加以区分。搜索节点的等待区间包含其前驱节点中所有可能执行的等待并移动动作的信息。扩展一个节点时，等待区间会向前投影至其所有后继节点，因此，关于可能的等待并移动动作的信息会从搜索树的根节点（起始节点）沿着所有分支传播。我们将实现这一原则的 SIPP 修改版本称为带（等待）区间投影的安全区间路径规划，即 SIPP-IP。在本文余下部分，我们将把 SIPP-IP 节点的等待区间称为时间区间（或简称区间）。提及图顶点的安全区间时，我们始终保留“安全”一词，以免混淆。

投影区间 投影操作的作用是将所有可用等待和移动动作的信息从前驱节点传播到后继节点。形式上，投影过程的输入是一个 SIPP-IP 搜索节点 $n = (v, [t_l, t_u])$，其中 $[t_l, t_u]$ 位于 v的某个安全区间内；另一个输入是图边 $e = (v, v')$，沿着该边传播区间。输出是时间区间集合 $TI = \{ti = [t', t'']\}$，使得：

- 每个结果区间都属于目标顶点的某个安全区间：$\forall ti_k \in TI\ \exists si \in SI(v') : ti_k \subseteq si$
- 结果区间彼此不重叠：$ti_k \cap ti_l = \emptyset, \forall k \neq l$
- 对于由 e 指定的任意转移，只要它从源区间中的任意时间步开始，与该转移终点对应的时间步

若该转移有效（执行时不会发生碰撞），则属于某个结果区间： $\forall t \in [t_l, t_u]$ s.t. $(e,\ t)$ 是无碰撞转移 $\exists \hat{t} \in TI$ s.t. $\hat{t} = t + w(e)$

区间投影操作的一般示例如图 3 所示。源图顶点以蓝色椭圆表示，目标顶点（出边到达的顶点）以绿色表示。黑色圆柱体表示顶点的阻塞区间。待投影的时间区间以黄色标出。投影得到的时间区间以青色、绿色和橙色显示。请注意，由于目标顶点中存在中间的不安全区间，投影后的时间区间首先被拆分为两个区间（落入不安全区间的粉色转移会被排除）。其次，虽然转移落入目标顶点的安全区间，但在转移过程中会导致碰撞的转移（灰色箭头）也会被剪除。因此，较低的时间区间会被裁剪，而较高的时间区间则会被拆分为两个区间。

投影操作的一种直接技术实现方式，是依次对构成输入区间的 e 时间步应用该操作，然后筛除无效转移，并将得到的时间步归并为区间。这类似于采用时间步的 A*算法；不过，A*算法得到的原子搜索状态会被压缩为 SIPP-IP 的区间状态，因此 SIPP-IP 的搜索树更加紧凑。事实上，根据碰撞检测机制的具体定义，还可以提出计算效率更高的区间投影操作实现方式。附录介绍了其中一种高效实现，它针对问题的基于网格的表示进行了专门设计，该表示最初见于（Cohen 等人，2019），并用于我们的实验中（每条边都关联着智能体执行运动时经过的一系列单元格）。[2]

为说明投影如何帮助求解标准 SIPP 无法求解的实例，回顾图 2 所示的示例。扩展起始节点时，SIPP-IP 会将 A 的时间区间（即 $[0, 5]$，与起始节点的安全区间重合）投影到后继节点上。这将得到 $[2, 7]$ 区间。扩展节点 $((B, vel = 1), [2, 7])$ 时，我们现在可以通过尝试在属于 $(B, vel = 1)$ 时间区间的任意时刻执行动作（再次通过投影操作），来应用匀速运动动作和减速动作。这样，B 到 C 的匀速运动动作将在 $t = 4$ 时执行，遵循 SIPP 尽早到达后继节点的原则，即在 $t = 5$ 时到达。等待区间的投影将得到 $[5, 8]$。最后，扩展 $((C, vel = 1), [5, 8])$ 时，将生成节点 $((D, vel = 0)\ [7, \infty))$。选择该节点进行扩展时，搜索将成功终止，并报告已找到解。

SIPP-IP 描述 SIPP-IP 首先根据给定的图顶点构建起始节点。起始区间最初仅包含作为输入提供的第一个时间步，，

[2]彩色摘录以及附录可能不会出现在稿件的会议版本中。

算法1： SIPP-IP

函数 findPath($v_{start}, t_{start}, v_{goal}, G(V, E)$ SI):
1 OPEN $\leftarrow \phi$, CLOSED $\leftarrow \phi$
2 $ti = [t_{start}, t_{start}]$
3 **如果** $v_{start}.vel = 0$, **则**
4 $ti.t_u \leftarrow$ SI的上界（v_{start}, ti）
5 $n_{start} \leftarrow$ （v_{start}, ti）
6 f（n_{start}） $\leftarrow t_{start} + h$（v_{start}）
7 添加 n_{start} 到 OPEN
8 **当** $OPEN \neq \phi$ **时执行**
9 $n \leftarrow$ 从 OPEN 中取出 f-值最小的状态
10 将 n 从 OPEN 中移除，将 n插入 CLOSED
11 **如果** $n.v = v_{goal}$ **那么**
12 **返回** $\pi \leftarrow$ 重建路径(n, n_{start})
13 后继节点 $\leftarrow$getSuccessors(n, $G(V$, $E)$ SI)
14 **对于**每个 n'，在 *succ* **执行**
15 **如果** n' 在 *CLOSED* 或 *OPEN* 中 **则**
16 **继续**
17 $f(n') \leftarrow n'.t_l + h(n'.v)$
18 将 n' 添加到 OPEN
19 **返回** ϕ

函数 getSuccessors(n, $G(V$, $E)$ SI):
20 SUCC = ϕ
21 **对** 每个 $e = (n.v, v')$ 可用运动 **执行**
22 $intrvls$ = projectIntervals(n, e, SI)
23 **如果** $v'.vel = 0$**则**
24 **对** 每个 ti在 $intrvls$ **执行**
25 $ti.t_u \leftarrow$ SI的上界（v', ti）
26 **对于** ti 中的 $intrvls$，**执行**
27 将（v', ti）插入 SUCC
28 **返回** SUCC

如果起始顶点允许等待，即智能体的初始配置的速度为零，则将该节点区间的上界扩展至安全区间（起始顶点的安全区间）的上界，t_{start} 位于该区间内。

随后，SIPP-IP 遵循 SIPP/A*算法的一般流程。每次迭代时，它首先从 $OPEN$ 中选择最佳节点，即 f 值最小的节点。SIPP-IP 节点 $n =$（v, $[t_l, t_u]$）的 f 值定义为 f（n） $= g$（n） $+h$（n） $= t_l + h$（v）。与 SIPP 类似，该节点的 g 值是智能体到达某个 n的最早可能时间步，即 t_l。h 值与时间区间无关，且针对图顶点定义。它用于估计从当前顶点到目标顶点的行进时间（例如，等于顶点间的直线距离除以智能体的最大速度）。与 SIPP 中一样，我们假设启发式函数对于 SIPP-IP 是可采纳且一致的。

选定一个节点后，会生成其后继节点。对于每条出边，我们都应用上述投影操作。因此，对于每条边，我们会得到一组

投影时间区间。如果目标构型允许等待（速度为零），那么我们会将投影区间的上界扩展至目标顶点对应安全区间的上界。因此，每个生成的后继节点都通过其时间区间隐含地包含了有关从前驱节点出发的所有可能转移，以及在当前节点执行的所有可能等待动作的信息。

最后，将每个生成的后继节点插入 $OPEN$，前提是它尚未存在于搜索树中。当从 $OPEN$ 中取出与目标顶点对应的节点时，算法停止。此时可以重建路径。为此，我们从目标节点开始反向遍历，并在每次迭代中执行以下操作。设 n 为当前节点（初始设为目标节点 n_{goal}），t 为时间变量（初始值等于 $n_{goal}.t_l$），n_{parent} 为 n 的父节点，e 为它们之间的转移。如果智能体可以在 n 等待，则将元组 $(e,\ n.t_l - cost(e))$ 添加到 π，并修改 $n = n_{parent}$ 和 $t = n.t_l - cost(e)$。如果智能体不能在 n 等待，则将 $(e,\ t - cost(e))$ 添加到 π，并修改 $n = n_{parent}$ 和 $t = t - cost(e)$。重复此过程，直到到达起始节点。如果最终值 t 与 t_{start} 不匹配，则智能体必须在起点等待（等待 $t - t_{start}$ 个时间步）。

SIPP-IP 的伪代码见算法1。

SIPP-IP的理论性质

根据 SIPP-IP 状态的定义，我们可以将每个 SIPP-IP 状态视为一系列带时间步的A*（A*-TS）状态。也就是说，$n_{SIPP-IP}(v,\ [t_l, t_u]) = \{s_{A*-TS}(v,\ t) : t \in [t_l, t_u]\}$。在接下来的证明中，我们将使用这一关系。

定理1。 *SIPP-IP* 是完备的。

证明。 在所提出的函数 *getSuccessors* 中，我们尝试使用输入顶点上的所有可用边。随后，对于每条边，我们使用函数 *projectIntervals* 获取相应区间；这些区间包含所有可能的有效时间步，使得从输入状态时间区间内的某个时间步出发，能够到达该边的目标节点（根据*project-Intervals*的定义）。这等价于通过一条边，从输入状态所包含的 A*-TS 状态生成所有 A*-TS 状态。使用所有边后，我们便生成了通过“移动”动作能够生成的所有 A*-TS 状态。紧接着，我们检查目标顶点处是否可以执行等待动作；如果可以，就将所得时间区间扩展至目标顶点所在安全区间的上界。这等价于对目标顶点处的所有 A*-TS 状态执行等待动作。因此，在 *getSuccessors* 中会生成所有有效的 A*-TS 后继节点（并将其封装在 SIPP-IP 状态中）。由此可见，SIPP-IP 可以视为 A*-TS 的一种改进版本：每次迭代中，它会同时扩展多个状态，生成这些状态的所有后继节点，并将其插入 OPEN（根据定义，所有生成的 A*-TS 状态都会无损地转化为 SIPP-IP 状态）。因此，SIPP-IP 始终会生成并扩展所有 A*-TS 状态；由于 A*-TS 是完备的，SIPP-IP 也完备。

□

引理 1。 *SIPP-IP* 状态 $n_{SIPP-IP}(v, [t_l, t_u])$ 中具有最小 f 值的 *A*-TS* 状态为 $n_{A*-TS}(v, t_l)$。

证明。 回想一下，A*-TS 状态的 f 值等于$n.t + h(n.v)$。由于同一个 SIPP-IP 状态中所有 A*-TS 状态的 v 都相同，因此它们的 h 值相等。因此，时间最小的状态，即 t_l，是 f 值最小的状态。 □

定理 2。 *SIPP-IP* 是最优的。

证明。 由于 SIPP-IP 是完备算法，且代价（时间）被纳入状态标识符，因此可以保证最优状态会被扩展。因此，只需证明第一个包含目标顶点的扩展状态是最优状态，即其中包含时间最优（最小）的 A*-TS 状态。再次将 SIPP-IP 状态视为一系列提取出的 A*-TS 状态。设第一个包含目标顶点的扩展 SIPP-IP 状态为$n_{SIPP-IP}(v_{goal}, [t_l, t_u])$，其中包含 A*-TS 状态$n_{A*-TS}(v_{goal},\ t_l)$。假设 n_{A*-TS}不是最优的 A*-TS 状态，但 OPEN $f(n')$ 中存在另一个状态 n'_{A*-TS}，其 f 值最小，且小于 $f(n)$。根据引理 1，n'_{A*-TS}位于 OPEN 中某个 SIPP-IP 状态 $n'_{SIPP-IP}$的底部，即$n'_{A*-TS}.t = n'_{SIPP-IP}.t_l$。由于 SIPP-IP 中的状态按照值 $f(n_{SIPP-IP}) = n.t_l + h(n.v)$ 排序，而该值等于底部 A*-TS 状态 $f(n_{A*-TS})$ 的 f 值，因此状态 $n'_{SIPP-IP}$本应先于状态 $n_{SIPP-IP}$扩展，因为 $f(n'_{A*-TS}) < f(n_{A*-TS})$，这就产生了矛盾。因此，不存在 f 值小于 $f(n_{A*-TS})$ 的 A*-TS 状态，所以 $n_{SIPP-IP}$ 是最优状态。

□

实证评估

我们在实验中使用了 MovingAI 基准测试（Sturtevant 2012）中的五种不同大小和拓扑结构的地图：空地图（大小为 64x64）、房间地图（64x64）、仓库地图（84x170）、随机地图（128x128）和悉尼地图（256x256）。每张地图上都逐步增加移动障碍物（MO）的数量；对于每种障碍物数量，我们生成了 200 个不同的实例，其移动障碍物的轨迹各不相同。这些轨迹通过随机分配每个移动障碍物的起点和目标位置生成，并让其以随机速度从起点移动到目标。此外，移动障碍物可以在任意单元格等待随机数量的时间步。对于每张地图，我们生成了移动障碍物密度分别为 1/25、1/20、1/15、1/10、1/5、1/4 和 1/3 的实例，其中密度是移动障碍物数量与空闲格子数量之比。智能体的起点和目标位置分别固定在地图的左上角和右下角。请注意，我们的数据集中有许多实例（尤其是移动障碍物密度较高的实例）通常无法求解。

将智能体建模为一个圆盘，其直径等于网格单元的边长。构型定义为 $(x,\ y,\ \theta,\ vel)$，其中 $x,\ y$ 是该单元格的坐标，$\theta \in$

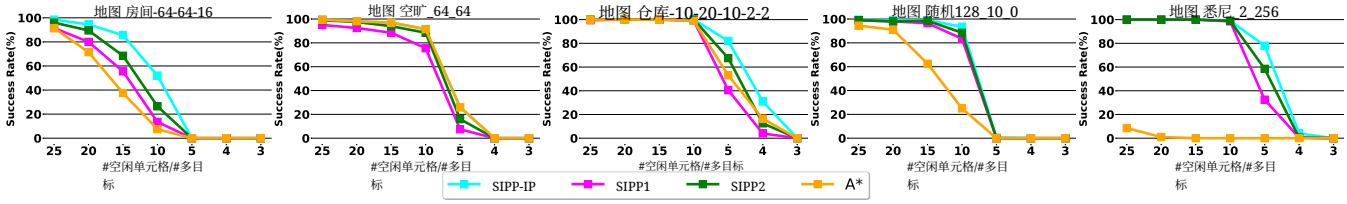

图4：评估算法的成功率。

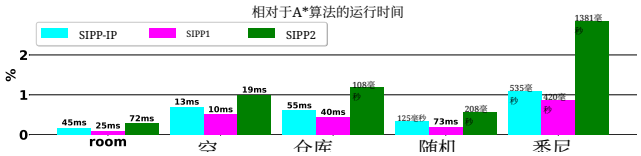

图5：SIPP-IP、SIPP1 和 SIPP2 与 A*算法的运行时间对比。柱形上方标出了绝对值（单位：毫秒）。

	room	空	仓库	随机	悉尼	因子
SIPP1	93%	82%	69%	89%	97%	$>$
SIPP2	90%	79%	66%	85%	92%	
SIPP1	77%	46%	16%	55%	30%	$>5\%$
SIPP2	60%	39%	11%	44%	20%	
SIPP1	42%	7%	1%	7%	4%	$>50\%$
SIPP2	22%	3%	0%	1%	3%	

表 1：与 SIPP-IP 相比，SIPP1/SIPP2 解的代价更高的百分比。

$\{0°, 90°, 180°, 270°\}$ 是朝向，$vel \in \{0, 2\}$ 是智能体的速度。

定义了以下运动学动力学运动基元：加速、减速、匀速。加速（减速）基本动作：从速度为零（最大速度为 2cell/s）的构型出发，沿初始朝向以固定加速度 0.5cell/s^2 (-0.5cell/s^2)运动，直至达到最大速度（速度为零）。智能体以这种方式移动 4 个单元格，且朝向不变。匀速运动基元：以最大速度向前移动一个单元格，即初始速度和最终速度均为 2cell/s，朝向保持不变，但单元格会根据朝向改变一个。除此之外，我们还考虑了原地旋转和等待动作。当速度为零时，智能体可以在 2 秒内旋转 90°，或等待一个时间步。时间步设为 0.1 秒。

碰撞检测采用保守策略。我们禁止智能体与任何多目标在同一时间接触同一个单元格。因此，当多目标哪怕只是部分接触某个单元格时，该单元格的阻塞区间就会为该多目标预留。只有当运动基元不会导致智能体在阻塞区间内接触某个单元格时，智能体才可以使用该运动基元。此外，如果智能体接触某个单元格的时间范围端点不是整数，则会以更安全的方式将其扩展到最接近的整数。

我们将 SIPP-IP 与采用时间步的 A*算法以及 SIPP 的两种直接扩展（SIPP1 和 SIPP2，这两种扩展均不完备）进行了比较。SIPP1 是对 SIPP 的一种修改，它只为速度不为零的构型生成运动学动力学可行后继节点。SIPP2 进一步允许重新扩展搜索节点（这相当于允许在同一安全区间内的不同时间步到达速度非零的构型）。实现 SIPP1/SIPP2 时，我们采用了 SIPPwRT（Ma 等，2019）中的技术，因此后者可视为已纳入比较。所有规划器的 C++ 源代码均已公开提供。[3]

实验在一台配备英特尔酷睿 i7-10700F 处理器 @ 2.90GHz $\times$ 16 和 32Gb 内存的 PC 上进行。我们为所有算法设定了生成节点上限：100,000,000。对于每个实例，我们记录算法是否找到了解，并记录算法的运行时间和解的代价。

图 4 展示了成功率（SR）曲线，其中 SR 是成功求解的实例数占所有测试实例总数的比例。事实上，所有与 SIPP-IP 竞争的方法的 SR 都低于 SIPP-IP（空地图除外，在该地图上 A*算法与 SIPP-IP 的 SR 相同）。对于 SIPP1 和 SIPP2，这可以用它们的不完备性来解释。理论上，A*算法的 SR 应始终与 SIPP-IP 相同，但在实践中，A*算法的搜索空间很大，许多次运行中生成节点数都超过了设定的 100,000,000 个上限。

对于每张地图，我们还分析了 SIPP-IP、SIPP1 和 SIPP2 成功求解的实例的代价。结果见表 1。表格中的每个单元格都表示，在多少个实例中（按百分比计），SIPP1/SIPP2 解的代价比 SIPP-IP 找到的最优解的代价高出一个固定倍数。显然，在大多数实例中，竞争方法的代价都高于 SIPP-IP；对于某些地图（例如房间），其代价显著超过 SIPP-IP（超过 50%）的实例比例相当高，最高达 42%。

最后，算法的运行时间分析如图 5 所示。事实上，所有版本的 SIPP（包括完备版本 SIPP-IP）都明显快于 A*算法，计算时间减少了两个数量级。此外，SIPP-IP 比 SIPP2 更快，但仍不及 SIPP1。这是预期之中的，因为 SIPP1 采用了直接的扩展策略，而这种策略会遗漏许多 SIPP-IP 能够生成的后继节点。SIPP-IP 的运行时间绝对值（平均值）表明，除最大地图外，其在所有地图上的运行时间都不到 0.1 秒；这说明所提出的规划器可用于真实机器人系统，在这些系统中，考虑运动学与动力学约束可能至关重要。

[3] https://github.com/PathPlanning/SIPP-IP

结论

本文提出了一种经过证明完备且最优的知名安全区间路径规划算法变体，能够处理运动学与动力学约束。我们证明，在所考虑的设定下，扩展 SIPP 的直接方法均告失败，因此需要采用更复杂的算法。随后，我们提出了该算法，并从理论和实证方面进行了分析。未来的研究方向包括将所提出的算法嵌入多智能体路径规划求解器，并在真实机器人上开展实验。

致谢

本研究部分得到了俄罗斯联邦政府分析中心依据补助协议（协议标识符 00000D730321P5Q0002）以及与莫斯科物理技术学院于2021年11月1日签订的第70-2021-00138号协议所提供的人工智能领域研究中心资助。

参考文献

Ali, Z. A.; 和 Yakovlev, K. 2021. 面向具有运动学约束的多智能体路径搜索的优先级 SIPP。载于 Ronzhin, A.; Rigoll, G.; 和 Meshcheryakov, R. 编，《交互式协作机器人》，1–13。Cham：Springer 国际出版。ISBN 978-3-030-87725-5。Cohen, L.; Uras, T.; Kumar, T. S.; 和 Koenig, S. 2019. 多智能体运动规划的最优与有界次优方法。载于《第*12*届年度组合搜索研讨会论文集（*SOCS 2019*）》，44–51。Hart, P. E.; Nilsson, N. J.; 和 Raphael, B. 1968年。关于启发式确定最小代价路径的形式基础。《*IEEE*系统科学与控制论汇刊》，4(2)：100–107。Li, J.; Chen, Z.; Harabor, D.; Stuckey, P. J.; 和 Koenig, S. 2022年。MAPF-LNS2：通过大邻域搜索快速修复多智能体路径搜索问题。载于《*AAAI*人工智能会议论文集》。Ma, H.; H¨onig, W.; Kumar, T. K. S.; Ayanian, N.; 和 Koenig, S. 2019. 具有运动学约束的多智能体取送终身路径规划。载于《第*33*届*AAAI*人工智能会议论文集（*AAAI 2019*）》，7651–7658。Narayanan, V.; Phillips, M.; 和 Likhachev, M. 2012. 面向动态环境的任意时刻安全区间路径规划。载于《*2012*年*IEEE/RSJ*智能机器人与系统国际会议论文集（*IROS 2012*）》，4708–4715。Phillips, M.; 和 Likhachev, M. 2011. SIPP：面向动态环境的安全区间路径规划。载于《*2011*年*IEEE*机器人与自动化国际会议论文集（*ICRA 2011*）》，5628–5635。Pivtoraiko, M.; 和 Kelly, A. 2011. 基于状态格运动基元的运动规划与动力学约束。载于《*2011*年*IEEE/RSJ*智能机器人与系统国际会议》，2172–2179。IEEE。Sturtevant，N. R. 2012。《基于网格的寻路》基准测试。*IEEE*《计算智能与电子游戏汇刊》，4(2)：144–148。Yakovlev，K.；Andreychuk，A. 2021。面向动态障碍物的时间最优任意角度路径规划。载于《第*31*届国际自动规划与调度会议论文集》（*ICAPS 2021*），405–414。Yakovlev，K.；Andreychuk，A.；Stern，R. 2020。重新审视有界次优安全区间路径规划。载于《第*30*届国际自动规划与调度会议论文集》（*ICAPS 2020*），300–304。Yakovlev，K.；Andreychuk，A.；Vorobyev，V. 2019。面向差速驱动机器人的优先级多智能体路径搜索。载于《*2019*年欧洲移动机器人会议论文集》（*ECMR 2019*），1–6。

附录

环境的网格表示。网格常用于各类应用（机器人学、电子游戏等）中的环境表示。在对 SIPP-IP 的评估中，我们也使用了网格。具体而言，环境由一组网格单元 $\mathcal{C}$ 表示，其中每个单元格可以因静态障碍物而可通行或被阻挡。输入图 $G=$（即指定智能体可能的配置以及它们之间转移的图）中的每个顶点 v（V，E）现在还关联着一组单元格，这些单元格是智能体处于构型 v 时所接触的单元格。连接顶点 v 与顶点 v' 的每条边 e（v，v'）$\in E$ 都关联着一列（序列）单元格（c_0^e，c_1^e，.....），智能体执行运动 e 时会扫过这些单元格。此外，还为每个单元格 c_i^e 给出时间区间 $[lb_i^e, ub_i^e]$，表示智能体执行运动基元期间（相对于运动的起始时间）扫过该单元格的时间。边 $cost$（e）的代价也已给出，其定义为运动基元的持续时间。

移动障碍物产生的安全/非安全区间现在与网格单元相关联。也就是说，对于每个单元格 c，给定一个有限的（互不重叠的）时间区间序列 SI^c，即安全区间序列，在这些区间内没有障碍物扫过该单元格。为统一处理，我们将静态障碍物占据的单元格的 SI^c 赋为 $\emptyset$。对安全区间求补集，即可定义单元格的碰撞区间。因此，图顶点$v \in V$的安全区间可定义为智能体处于 v时所占据的各单元格安全区间的交集。

接下来，我们介绍如何针对网格高效实现*ProjectIntervals*过程。

在网格中投影时间区间。请看图 6 所示的示例。假设有一个 SIPP-IP 状态n =(v，$[2,17]$)。此外，假设智能体处于 v 时只接触一个单元格。[4] 考虑一个运动基本动作 e(v，v_{end}) =(c_0^e，c_1^e，c_2^e)：$[lb_0^e = 0, ub_0^e = 3]$，$[lb_1^e = 2, ub_1^e = 4]$，$[lb_2^e = 3, ub_2^e = 5]$ 这意味着，执行基本动作 e时，智能体从时间步 0 到时间步 3 一直扫过单元格 c_0^e。它还会在时间步 2 首次接触单元格c_1^e，并继续扫过该单元格 2 个时间步；在时间步 3 首次接触单元格 c_2^e，并同样继续扫过 c_2^e 2 个时间步。请注意，智能体可能会同时扫过多个网格单元

同一时间步，这符合智能体具有身体/形状这一隐含假设。该运动基元的代价为 5，这意味着智能体将在移动开始后的 5 个时间步后到达最终构型，v_{end}

任务是找出包含所有时间步的时间区间，在这些时间步中，智能体可以通过执行e到达 v_{end} $[2,17]$。显然，最直接的方法是模拟智能体从源区间 $[2,17]$ 中的每个时间步出发进行移动，并计算经过（c_0^e，c_1^e，c_2^e）后哪些时间步仍然有效。然后便可根据这些时间步重建时间区间

[4]然而，我们即将介绍的投影过程与智能体在任意构型下接触多少个单元格无关。

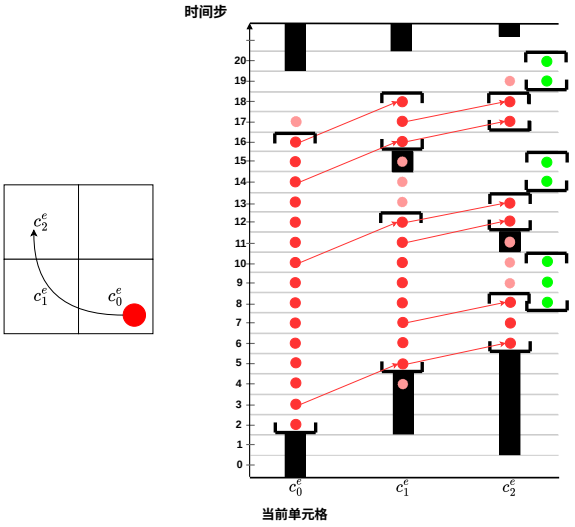

图 6：左图：一个扫过 3 个网格单元的运动基元。右图：将时间区间依次投影到构成该运动基元的网格单元上。

这些时间步。

不过，我们可以更高效地计算所得的时间区间。利用基本动作的序列表示，我们可以将初始时间区间依次投影到构成该基本动作的各个单元格上，直到到达最终单元。在此过程中，只需处理时间区间的端点（无需处理构成时间区间的所有时间步）。在每一步逐单元格进行投影时，我们需要得到这样的区间：1）包含智能体首次接触目标单元格时的所有时间步；2）仅包含在给定基本动作下可继续移动的有效时间步。第二个条件是指，从这些区间内的任意时间步开始执行该基本动作，都不会与所考察单元格中的任何非安全区间发生碰撞。为了验证基本动作的起始单元格满足第二个条件，我们也会将初始时间区间投影到该单元格上（然后再投影到下一个单元格）。

总的来说，完成一系列单元间投影后，我们会计算移动中最终单元（在我们的示例中为 c_2^e）的时间区间，也就是智能体首次触及该单元的时间。最后，为了得到智能体完成移动的时间步，我们应将这些时间步平移到基本动作的末尾。

在所考虑的示例中（回顾图 6），我们首先将区间 $[2,17]$ 投影到单元格 c_0^e 上，得到区间 $[2,16]$。请注意，后者与初始区间并不重合。原因是时间步 17 并非开始执行基本动作的有效时间步。如果移动在 $t=17$ 开始，那么在 $t=20$ 时，智能体仍会触及 c_0^e（根据 e，c_0^e 的扫掠区间持续 3 个时间步）。然而，这不符合 c_0^e 给定的安全区间。

接下来，将其投影到下一个单元格 c_1^e 后，得到两个相交区间——

算法 2： 投影区间

函数 projectIntervals(n, $e=(n.v,\ v')$ SI)**：**

1 $time_ints \leftarrow \{[n.t_l, n.t_u]\}$
2 $t \leftarrow 0$
3 **对** 每个 $cell$ 依次执行 $e.cells$ **do**
4 $new_ints \leftarrow \phi$
5 $\Delta \leftarrow lb^e_{cell} - t$
6 $t \leftarrow lb^e_{cell}$
7 **对**每个 ti 执行 $time\ ints$ **do**
8 **对** 每个 si 执行 $SafeIntervals(cell)$ **do**
9 $t_{earliest} \leftarrow max(ti.t_l + \Delta,\ lb(si))$
10 $t_{latest} \leftarrow min(ti.t_u + \Delta,$ $ub(si)\ -(ub^e{}_{cell} - lb^e_{cell}))$
11 **如果** $t_{earliest} \le t_{latest}$ **那么**
12 插入 $[t_{earliest}, t_{latest}] \rightarrow new_ints$
13 $time_ints \leftarrow new_ints$
14 $succ \leftarrow \phi$
15 **对于**每个 ti在 $time\ ints$ **执行**
16 $last\ cell \leftarrow e$中的最后一个单元格
17 $\Delta \leftarrow cost\ (e)\ - lb^e_{last_cell}$
18 $ti.t_l \leftarrow ti.t_l + \Delta$
19 $ti.t_u \leftarrow ti.t_u + \Delta$
20 插入 $(v',\ ti) \rightarrow succ$
21 **返回** $succ$

第一个安全区间内的区间 $[5,12]$ 和第二个安全区间内的区间 $[16,18]$。这意味着，智能体可以在时间步 $t \in [5,12] \cup [16,18]$首次接触 c^e_1。由于根据基本动作 e，智能体首次接触单元格 c^e_1 后还会继续扫过该单元格 2 个时间步，因此不允许在时间步 13 和 14 到达；若在时间步 13 或 14 到达，智能体就会与阻塞区间发生碰撞，因为时间步 15 在 c^e_1 处被阻塞。接下来，将区间 $[5,12]$ 投影后，在 c^e_2 的第一个安全区间内得到区间 $[6,8]$，在第二个安全区间内得到区间 $[12,13]$。同样，将区间 $[16,18]$ 投影到下一个单元格后，会在 c^e_2的第二个安全区间内得到区间 $[17,18]$。现在，我们得到了从输入状态出发、能够安全（经过所有单元格）到达单元格 c^e_2 时的所有时间步。假定函数返回的是基本动作结束时的时间步（而非触及最后一个单元格时的时间步），因此最后只需将得到的时间步向后平移相应的持续时间（$cost(e) - lb_{c^e_2}$）$= 2$，这样得到的区间便为$[8,10],[14,15]$ 和 $[19,20]$。

伪代码 算法首先在第 1 行，根据输入时间区间初始化得到的时间区间 $time\ ints$。然后，我们从初始单元格开始，依次遍历给定基本动作中的所有单元格。在每次迭代中，我们首先初始化一个新的空时间区间集合 $new\ ints$，用于存放当前单元格 $cell$处得到的投影区间。接着，我们计算从前一个单元格到达 $cell$ 所需的时间 Δ；如果是基本动作中的第一个单元格，则计算从输入顶点或该单元格到达所需的时间（使用辅助变量）。

顶点。随后，我们将当前存储在$time\ ints$ 中的已获得区间投影到 $cell$上。具体而言，对于$time\ ints$中的每个区间，我们在 $cell$ 的每个安全区间内找出最早时间步 $t_{earliest}$ 和最晚时间步 t_{latest}，即可完成此过程。根据输入时间区间 ti 和进行投影的安全区间 si 的端点，可以轻松计算这些时间步，如第 9、10 行所示。然后，如果所得时间步构成有效区间（第 11 行的条件），就将其加入 $new\ ints$。每次迭代结束时，我们将新的投影区间 $new\ ints$ 赋给 $time\ ints$，以便在下一个单元格上再次进行投影。所有迭代结束后，我们便得到此刻首次接触 e中最后一个单元格时对应的所有时间步构成的时间区间。最后，将所得时间区间平移至基本动作的末尾（即按值 $cost\ (e) - lb^e_{last\ cell}$进行平移），如第 18、19 行所示。之后，我们从 e的终点顶点构造后继状态 v'，每个后继状态对应一个所得时间区间，并将其返回。