Safe Interval Path Planning with Kinodynamic Constraints

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

[1] Moscow Institute of Physics and Technology, Moscow, Russia
[2] Federal Research Center for Computer Science and Control of Russian Academy of Sciences, Moscow, Russia
[3] AIRI, Moscow, Russia
ali.za@phystech.edu, yakovlev@isa.ru

Abstract

Safe Interval Path Planning (SIPP) is a powerful algorithm for solving a single-agent pathfinding problem where the agent is confined to a graph and certain vertices/edges of this graph are blocked at certain time intervals due to dynamic obstacles that populate the environment. The original SIPP algorithm relies on the assumption that the agent is able to stop instantaneously. However, this assumption often does not hold in practice, e.g. a mobile robot moving at a cruising speed cannot stop immediately but rather requires gradual deceleration to a full stop that takes time. In other words, the robot is subject to kinodynamic constraints. Unfortunately, as we show in this work, in such a case, the original SIPP is incomplete. To this end, we introduce a novel variant of SIPP that is provably complete and optimal for planning with acceleration/deceleration. In the experimental evaluation, we show that the key property of the original SIPP still holds for the modified version: it performs much fewer expansions compared to A* and, as a result, is notably faster.

Introduction

Planning a path in the presence of both static and moving obstacles is a challenging problem with topical applications in robotics, video games, and other domains. When the environment is fully observable and the trajectories of the dynamic obstacles are known (e.g., are predicted by the robot's perception system or by a global observer), it is reasonable to account for them while planning. A prominent method that is tailored for such a setting is Safe Interval Path Planning (SIPP) (Phillips and Likhachev 2011). This is a search-based algorithm that operates on a graph where vertices correspond to the configurations of the agent (e.g., position, heading, velocity, etc.) and edges correspond to the transitions between them. Due to the dynamic obstacles, certain vertices/edges of this graph are blocked at certain time intervals. SIPP accounts for that and constructs plans in which an agent can wait at the vertices to avoid collisions. An explicit assumption of SIPP is that the agent can start/stop moving instantaneously. Under this assumption, the algorithm is provably complete and optimal (w.r.t. the given spatial and temporal discretization).

Copyright © 2023, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

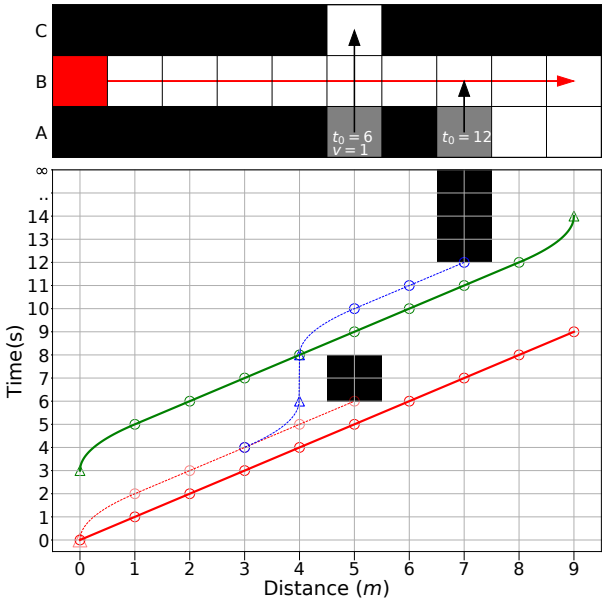

Figure 1: An example of the challenging path planning instance with dynamic obstacles.

In our work, we adopt an assumption (more realistic from the practical perspective) that decelerating to a full stop and, similarly, accelerating from the full stop takes time. In other words, the agent is subject to kinodynamic constraints. This makes SIPP inapplicable as shown in Fig. 1. In this example, the environment is discretized to the 3×10 grid, where the distance between the centers of the grid cells is $1\,\mathrm{m}$, and the agent has to go from the leftmost cell $B0$ to the rightmost one $B9$. Moreover, two dynamic obstacles are present in the environment that block cells $B5$ and $B7$ during the intervals shown in black in the bottom part of the figure. Assume that the agent's cruising speed is $1\,\mathrm{m\,s^{-1}}$. At this speed, it takes $1\,\mathrm{s}$ to move between the cells. Furthermore, the maximum acceleration/deceleration of the agent is $0.5\,\mathrm{m\,s^{-2}}$, which means that accelerating to the cruising speed from the full stop takes $2\,\mathrm{s}$ and $1\,\mathrm{m}$. Therefore, when starting to move from the full stop, the agent will arrive at the neighboring cell not in $1\,\mathrm{s}$, but rather in $2\,\mathrm{s}$. If the algorithm ignores that, like the original SIPP, which assumes

instantaneous acceleration, then the constructed plan, shown in red, will actually lead to the collision (if it is applied with real kinodynamic constraints) as shown by the dashed red line. A straightforward modification of SIPP that takes accelerating/decelerating actions into account is to apply only dynamically valid transitions, i.e., to prevent the wait in the states where the velocity is not zero. However, this variant will report *failure* to find the solution while producing a partial plan shown in blue. In this plan, the agent successfully stops before the first obstacle and waits for 2 s, but then is unable to pass the second obstacle as it is running late to arrive at $B7$. There is, however, a collision-free plan. It is shown in green. The reason why SIPP fails to find it is that, intuitively, SIPP postpones the wait actions and cannot reason about the consequences of waiting in different states of the search tree. This does not violate the theoretical guarantees when waiting is available in any state. On the other hand, when waiting is not always readily available, i.e., the agent needs time to stop, the algorithm becomes evidently incomplete. We will elaborate on this and provide technical details in the following sections of the paper.

To the best of our knowledge, no works on SIPP exist that directly address this issue. As a result, all known SIPP-based algorithms are incomplete in the setting where kinodynamic constraints of the agent have to be taken into account when planning. Our work fills this gap and presents an algorithm that is provably complete and optimal under such constraints: Safe Interval Path Planning with Interval Projection (SIPP-IP). To empirically evaluate it, we have conducted a wide range of experiments in which we compare SIPP-IP to several baselines that include other (non-complete) variants of SIPP one may think of and A*. Empirical results clearly show that SIPP-IP outperforms them all, as it is able to solve instances that are unsolvable by other planners and its runtime is two orders of magnitude lower than the one of A*.

Related Work

Search-based planning with predictably moving obstacles can be straightforwardly implemented as A* (Hart, Nilsson, and Raphael 1968) with discretized time. Taking the time dimension into account, however, leads to significant growth of the search space, especially when fine-grained time discretization is needed. To this end, (Phillips and Likhachev 2011) introduced SIPP, based on the idea of compressing sequences of timesteps into the time intervals and searching over these intervals. SIPP is provably complete and optimal under several assumptions, including that the agent can start/stop moving instantaneously. Later, numerous variants of SIPP emerged, enhancing the original algorithm, e.g., any-angle SIPP (Yakovlev and Andreychuk 2021), anytime SIPP (Narayanan, Phillips, and Likhachev 2012), different variants of bounded-suboptimal SIPP (Yakovlev, Andreychuk, and Stern 2020). Moreover, SIPP and its modifications are widely used as building blocks of some of the state-of-the-art multi-agent pathfinding solvers (Cohen et al. 2019; Li et al. 2022). None of these variants consider kinodynamic constraints.

Next, we mention only the works that to some extent deal with taking the agent's kinematic and/or kinodynamic constraints into account. (Ma et al. 2019) introduced SIPPwRT that allowed for the planning with different velocities. Still, acceleration actions and effects were not considered in this work. Similarly, (Yakovlev, Andreychuk, and Vorobyev 2019) described any-angle variant of SIPP that supported non-uniform velocities. Interestingly, the authors evaluate their planner on real robots. To plan safe trajectories for them, they suggested inflating the sizes of the moving obstacles, which alternatively can be seen as extending the blocked time intervals of certain graph vertices/edges. This ad-hoc technique was shown to perform reasonably well in practice, but in general, it raises the question of how to choose the offset. For example, if one enlarges the blocked intervals in the setting described in the Introduction (recall Fig. 1) by 1 s, SIPP will not find a solution. In (Ali and Yakovlev 2021), accelerating actions were straightforwardly integrated into SIPP, which, again, makes the algorithm incomplete (as we show in this work). (Cohen et al. 2019) suggested a variant of SIPP for the problem setting that assumes arbitrary motion patterns. However, this was not the main focus of the paper. Consequently, no techniques were proposed to take special care of the accelerating motions, and the empirical evaluation considered only the motions that start and end with zero velocity. Overall, to the best of our knowledge, no SIPP variant existed prior to this work that would be provably complete and optimal when the assumption of the original SIPP that "the robot can start/stop instantaneously" does not hold.

Problem Statement

We assume a discretized timeline $T = 0, 1, ...$ The agent is associated with a graph $G = (V, E)$, with the start and goal vertices: $start, goal \in V$. A vertex of the graph corresponds to the state of the agent, alternatively known as the configuration, in which the agent can reside without colliding with the static obstacles. Each configuration explicitly encompasses the velocity, vel, of the agent. For example, it may be comprised of the agent's coordinates and orientation as well as of its velocity: $v = (x, y, \theta, vel)$. Configurations with the same position/orientation but different velocities, indeed, correspond to different vertices of V. Configurations with $vel = 0$ are of special interest, as the agent may stay put (wait) only in them.

An edge $e = (v, u) \in E$ represents a *motion primitive* (Pivtoraiko and Kelly 2011), a small kinodynamically feasible fragment of the agent's motion that transfers the agent from v to (distinct) u. We assume that for any $v \in V$, a finite set of such motions is available and each motion takes an integer number of timesteps. Practicality-wise, this assumption is mild, as any real-valued duration may be approximated with a certain precision and represented as an integer. The duration of the motion defines the weight of the edge, $w(e) \in \mathbb{N}$.

Based on the source/target velocity, each motion (edge) can be classified as either accelerating, decelerating, or uniform. In the latter case, the agent's velocity at the target configuration is the same as in the source one, while in the

first two cases, it changes (increases and decreases, respectively). The presence of accelerating/decelerating motions makes our problem distinguishable from the works that assumed instantaneous acceleration[1].

Besides the agent and the static obstacles, the environment is populated by a fixed number of moving obstacles. We assume that their trajectories are known and converted (by an auxiliary procedure) to the collision and safe intervals associated with the graph vertices and edges; i.e., for each vertex, a finite sequence of non-overlapping time intervals $SI(v)$ is given, which is the sequence of the *safe intervals* at which the agent can be configured at v without colliding with any moving obstacle. Similarly, safe intervals are defined for each edge, $SI(e)$. If the agent executes a move e at any timestep outside $SI(e)$, a collision with a moving obstacle occurs.

A (timed) path for the agent, or a trajectory, is a sequence $\pi = (e_1, t_1), (e_2, t_2), ...(e_L, t_L)$, where $t_i \in T$ is the time moment the motion defined by the edge e_i is started. The cost of the trajectory is the timestep when the agent reaches the final vertex, $cost(\pi) = e_L + w(e_L)$. A trajectory is *feasible* if the target vertex of e_i matches the source vertex of e_{i+1} and $t_{i+1} \geq t_i + w(e_i)$ (with $t_0 = 0$). Moreover if $t_{i+1} > t_i + w(e_i)$, then the velocity at the source of e_{i+1} must be zero.

A feasible trajectory can also be seen as a *plan* composed of the move and wait actions, where the move actions are defined by the graph edges and the wait actions might occur at the vertices where the agent's velocity is zero. The duration of the wait action occurring after the i-th move action is computed as $\delta = t_{i+1} - (t_i + w(e_i))$. As $w(e_i)$ is integer, δ is integer as well.

A collision between the trajectory π and the dynamic obstacles occurs *iff* either of the conditions occur: *i*) the agent starts executing some motion primitive e at the timestep which is outside of $SI(e)$; *ii*) according to π, the agent waits at a vertex v outside of $SI(v)$.

The problem now is to find a feasible collision-free trajectory π that transfers the agent from $start$ to $goal$ on a given G (with the annotated safe intervals of vertices/edges). In this work, we are interested in solving this problem optimally, i.e., in reaching the goal as early as possible.

Method

As our method relies on SIPP, which in turn relies on A* with timesteps, we first discuss the background and then delve into the details of the suggested approach. We assume that a reader is familiar with vanilla A* for static graphs.

Background

A* with time steps The straightforward way to solve the considered problem is to use A* for searching the state space whose nodes are the tuples $n = (v, t)$, where v is the graph vertex and t is the time step by which it is reached. When expanding a node, the successors are generated as follows. First, for each $e = (v, u)$, if e is feasible, the node $n_{move} = (u, t + w(e))$ is added to the successors. Indeed, if the application of a motion e at time step t results in a collision, the resulting node is discarded. Second, if v allows for the wait, then the node corresponding to the wait action of the minimal possible duration of one time step $n_{wait} = (v, t + 1)$ is also added to successors. Again, if the safe intervals of v do not cover the time step $t + 1$, this node is discarded.

Other parts of the search algorithm are exactly the same as in A*. Notably, the g-values of the nodes equal the time components of their identifiers. I.e., the g-value of the node $n = (v, t)$ equals t, as this is the minimal known estimate of the cost (w.r.t to the current iteration of the algorithm) form $start$ to v.

SIPP In many scenarios, A* with time steps generates numerous nodes of the form $(v, t), (v, t + 1), (v, t + 2)$, etc., which are created via the use of the atomic wait actions. This leads to the growth of the search tree and slows down the search. To this end, SIPP relies on the idea of compressing the sequential wait actions to reduce the number of the considered search nodes.

The search nodes of SIPP are identified as $n = (v, [lb_j, ub_j])$, where $[lb_j, ub_j]$ is the j-th safe interval of v. For each node, SIPP stores the *earliest arrival time*: $t \in [lb_j, ub_j]$. Whenever a lower-cost path to n is found, $t(n)$ is updated, and this value serves as the g-value of the node similarly to A* with time steps.

When expanding a node, SIPP does not generate any separate successor corresponding to the wait action. Instead, it generates only the successors that correspond to the "wait and move" actions that land into the neighboring configurations within their safe intervals. This means that to generate a successor, SIPP waits the minimal amount of time possible so that when the move to the new configuration is used, we arrive at the new safe interval as early as possible. For example, if SIPP performs a move from a node $n = (v, [5, 10])$ to a node $n' = (v', [15, 18])$ and the duration of that move is 5 and $t(n) = 7$, then $t(n')$ is 15, meaning that after reaching n at $t = 7$, the agent waits there for 3 time steps, starts moving at $t = 10$, and arrives to n' at $t = 15$. This move cannot be performed earlier than $t = 10$, as in this case, it will end outside the safe interval of n'.

SIPP is much faster than A* and is provably complete and optimal under the following assumptions: inertial effects are neglected, the agent can start/stop moving instantaneously, and, consequently, the wait actions are available at all configurations.

What if the wait actions are not always available The intrinsic SIPP assumption that the agent can wait at any configuration often does not hold in practice, as many mobile agents cannot start/stop instantaneously. To reflect this, the accelerating/decelerating motions should be introduced, and the velocity variable should be added to the agent's configuration (as done in our problem statement). In this setting, SIPP is incomplete.

[1]Please note that our problem formulation, as well as the suggested method, is agnostic to how the state variables, e.g. velocity, change while the agent is executing a motion primitive. We are interested in the values of the state variables only at the endpoints of a motion primitive.

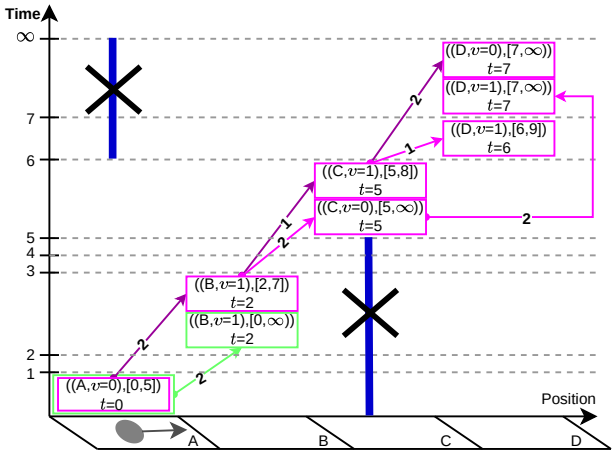

Figure 2: An example where the standard SIPP fails to find a solution, while SIPP-IP succeeds. The search tree of SIPP is shown in green. The search tree of SIPP-IP is indicated in purple.

Statement 1. *SIPP is incomplete when the agent is subject to kinodynamic constraints and wait actions are not available at any configuration.*

Proof. We prove by presenting an example for which SIPP is not able to find a solution whilst it exists. Consider a problem depicted in Fig. 2. The agent needs to reach cell D from cell A. The motion primitives are defined as follows:

- (accelerating motion) The agent starts moving from a cell with $vel = 0$ and ends with $vel = 1$ in the neighboring cell. The cost (duration) is 2 time steps.
- (uniform motion) The agent starts moving from a cell with $vel = 1$ and ends with $vel = 1$ in the neighboring cell. The cost (duration) is 1 time step.
- (decelerating motion) The agent starts moving from a cell with $vel = 1$ and ends with $vel = 0$ in the neighboring cell. The cost (duration) is 2 time steps.

Indeed, the agent can wait when $vel = 0$. The duration of the wait action is 1 time step. For the sake of brevity, we ignore the agent's orientation and motions that change it.

SIPP starts with expanding the initial search node $((A, vel = 0), [0, 5])$ in which only the accelerating motion is applicable. This results in generating the node $((B, vel = 1), [0, \infty))$ with the arrival time $t = 2$, shown in green in Fig. 2. The agent cannot wait in this configuration. Thus, it can only continue moving, arriving at C either at $t = 3$ using the uniform motion action or at $t = 4$ using the decelerated motion. Both moves are invalid, as they do not reach C within its safe interval which starts at 5. Thus, no successors are generated and no search nodes that can be expanded are left in the SIPP's search tree. The algorithm terminates, failing to report a solution. The latter, however, does exist: the agent needs to wait for 2 time steps at A and then continue moving towards D. In this way, it will arrive at C at $t = 5$ satisfying the safe interval constraint. □

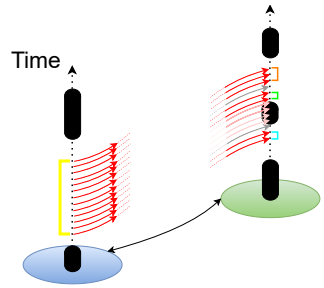

Figure 3: Projection of the time interval in SIPP-IP.

Next, we present a modification of SIPP that is complete and optimal under the considered assumptions.

SIPP-IP: Safe Interval Path Planning with (Wait) Interval Projection

Idea The main reason why standard SIPP fails to solve planning instances like the one presented above is because the information about possible wait actions is not propagated from predecessors to successors. In the considered example, when achieving the search node $((B, vel = 1), [0, \infty))$, SIPP "forgets" that the agent can wait in the predecessor and perform the move action at any time step until the end of the predecessor's safe interval. This is not a problem when the agent can wait at any configuration, but leads to incompleteness in the case we are considering.

To this end, we substitute the safe interval as the search node's identifier with another time interval which we refer to as the *wait interval*. For any search node, this interval belongs to the safe interval of the graph vertex. Indeed, nodes with the same vertex but different wait intervals should be distinguished. The wait interval of a search node incorporates information about all possible wait-and-move actions that can be performed in its predecessor. When a node is expanded, the waiting interval is *projected* forward to all of its successors; thus, the information about the possible wait-and-move actions is propagated from the root of the search tree (start node) along all of its branches. We call the modification of SIPP that implements this principle Safe Interval Path Planning With (Wait) Interval Projection, or SIPP-IP. Henceforth, we will refer to the wait interval of a SIPP-IP node as time interval (or, simply, interval). When talking about the safe intervals of the graph vertices, we will never omit "safe" to avoid confusion.

Projecting intervals The role of the projection operation is to propagate the information on all of the available wait-and-move actions from the predecessor to the successor. Formally, the input for the projection procedure is a SIPP-IP search node, $n = (v, [t_l, t_u])$, where $[t_l, t_u]$ resides inside one of the safe intervals of v, and a graph edge $e = (v, v')$ along which the interval should be propagated. The output is the set of time intervals $TI = \{ti = [t', t'']\}$, s.t.:

- each resultant interval belongs to one of the safe intervals of the target vertex: $\forall ti_k \in TI \; \exists si \in SI(v') : ti_k \subseteq si$
- resultant intervals do not overlap: $ti_k \cap ti_l = \emptyset, \forall k \neq l$

- for any transition, specified by e, that starts at any time step of the source interval, the time step corresponding to the end of this transition belongs to one of the resultant intervals if the transition is valid (no collisions occur when performing it): $\forall t \in [t_l, t_u]$ s.t. (e, t) is a collision-free transition $\exists \hat{t} \in TI$ s.t. $\hat{t} = t + w(e)$

A general example of how projecting interval operation works is depicted in Fig. 3. The source graph vertex is denoted by the blue oval, and the target vertex (where the outgoing edge lands) is shown in green. Black cylinders correspond to the blocked intervals of the vertices. The time interval to be projected is marked in yellow. The resultant time intervals are shown in cyan, green, and orange. Observe that due to the existence of the intermediate unsafe interval in the target vertex, the projected time interval is, first, split into the two ones (pink transitions landing in unsafe intervals are ruled out). Second, the transitions that land in the safe intervals of the destination vertex but that lead to a collision in the course of the transition (gray arrows) are also pruned. Thus, the lower time interval is trimmed, while the upper is split into two ones.

A straightforward technical implementation of the projecting operation may involve the sequential application of e to the time steps forming the input interval with further filtering out the invalid transitions and grouping the resultant time steps into the intervals. This resembles A* with time steps, but the resultant atomic search states of A* are compressed into the interval states of SIPP-IP. Thus, the search tree of SIPP-IP is more compact. Indeed, more computationally efficient implementations of the projection operation may be suggested, depending on how the collision detection mechanism is specified[2].

To demonstrate how projecting helps in solving instances that were unsolvable for standard SIPP, recall the example depicted in Fig. 2. When expanding the start node SIPP-IP projects, the time interval of A, which is $[0, 5]$ (coincides with the safe interval for the start node), to the successor. This results in $[2, 7]$ interval. When expanding the node $((B, vel = 1), [2, 7])$, we can now apply both uniform motion action and decelerating action by trying to commit them at *any* time step that belongs to the time interval of $(B, vel = 1)$ (via the projection operation, again). In such a way, the uniform motion action from B to C will be committed at $t = 4$, following the SIPP's principle of reaching the successor as early as possible, i.e. at $t = 5$. The projection of the interval of B will give us $[5, 8]$. Finally, when expanding $((C, vel = 1), [5, 8])$ the node $((D, vel = 0), [7, \infty))$ will be generated. When this node will be chosen for expansion, the search will report finding the solution.

SIPP-IP description SIPP-IP starts with forming the start node from the graph vertex. Initially, the start interval contains only the first time step, t_{start}, provided as input. If the start vertex allows for waiting, i.e., the velocity of the initial configuration is zero, the upper bound of the node's interval is extended to the upper bound of the safe interval (of the start vertex), in which t_{start} resides.

[2]See more in the pre-print available on arXiv.

Algorithm 1: SIPP-IP

Function `findPath`$(v_{start}, t_{start}, v_{goal}, G(V,E), SI)$**:**

OPEN $\leftarrow \phi$, CLOSED $\leftarrow \phi$
$ti = [t_{start}, t_{start}]$
if $v_{start}.vel = 0$ **then**
  $ti.t_u \leftarrow$ upper bound of $SI(v_{start}, ti)$
$n_{start} \leftarrow (v_{start}, ti)$
$f(n_{start}) \leftarrow t_{start} + h(v_{start})$
Add n_{start} to OPEN
while *OPEN* $\neq \phi$ **do**
  $n \leftarrow$ state from OPEN with minimal f-value
  remove n from OPEN, insert n to CLOSED
  if $n.v = v_{goal}$ **then**
    return $\pi \leftarrow$ ReconstructPath(n, n_{start})
  succ $\leftarrow$ getSuccessors$(n, G(V,E), SI)$
  for *each* n' *in succ* **do**
    if n' *in CLOSED or in OPEN* **then**
      continue
    $f(n') \leftarrow n'.t_l + h(n'.v)$
    Add n' to OPEN
return ϕ

Function `getSuccessors`$(n, G(V,E), SI)$**:**

SUCC = ϕ
for *each* $e = (n.v, v')$ *in available motions* **do**
  $intrvls$ = projectIntervals(n, e, SI)
  if $v'.vel = 0$ **then**
    for *each* ti *in* $intrvls$ **do**
      $ti.t_u \leftarrow$ upper bound of $SI(v', ti)$
  for ti *in* $intrvls$ **do**
    insert (v', ti) to SUCC
return SUCC

Then SIPP-IP follows the general outline of SIPP/A*. At each iteration, it, first, selects the best node from $OPEN$, i.e., the one with the minimal f-value. The f-value of a SIPP-IP node $n = (v, [t_l, t_u])$ is defined as $f(n) = g(n) + h(n) = t_l + h(v)$. Similarly to SIPP, the g-value of the node is the earliest time step the agent can arrive at n, that is t_l. The h-value is independent of the time interval and is defined for graph vertices. It estimates the travel time from the vertex to the goal (e.g., it equals the straight-line distance between the vertices divided by the maximum speed of the agent). As in SIPP, we assume the heuristic to be consistent.

After selecting a node, its successors are generated. For each outgoing edge, we apply the projecting operation as described above. As a result, for each edge, we obtain a set of projected time intervals. If the target configuration allows for waiting (the velocity is zero), then we extend the upper bounds of the projected intervals to the upper bounds of the corresponding safe intervals of the target vertex. As a result, each of the generated successors implicitly encompasses the information both about all possible transitions from the pre-

decessor and all possible wait actions in the current node.

Finally, each generated successor is inserted into $OPEN$ in case it is not already present in the search tree. The algorithm stops when a node corresponding to the goal vertex is extracted from $OPEN$. At this stage, the path can be reconstructed. To do so, we go backward from the goal node and, at each iteration, do the following. Let n be the current node (initially set to the goal node, n_{goal}), t – the time variable (initially equal to $n_{goal}.t_l$), n_{parent} – parent of n, and e the transition between them. If the agent can wait at n, then we add the tuple $(e, n.t_l - cost(e))$ to π and change $n = n_{parent}$ and $t = n.t_l - cost(e)$. If the agent cannot wait at n, we add $(e, t - cost(e))$ to π and change $n = n_{parent}$ and $t = t - cost(e)$. This is repeated until we reach the start node. If the final value t does not match t_{start}, then the agent has to wait at the start (for $t - t_{start}$ time steps).

The pseudo-code of SIPP-IP is presented in Algorithm 1.

Theoretical Properties of SIPP-IP

According to the definition of the SIPP-IP state, we can view each SIPP-IP state as a sequence of A*-with-Time-Steps (A*-TS) states. That is $n_{SIPP-IP}(v, [t_l, t_u]) = \{s_{A*-TS}(v, t) : t \in [t_l, t_u]\}$. We will use this relation along in the next proofs.

Theorem 1. *SIPP-IP is complete.*

Proof. In the presented function *getSuccessors*, we try to use all available edges on the input vertex. Then, for each edge using the function *projectIntervals*, we get the intervals that contain all possible valid timesteps at which we can get at the target node of the edge starting from a timestep in the time interval of the input state (by definition of *projectIntervals*). This is equivalent to generating all A*-TS states from the A*-TS states included in the input state using an edge. By using all the edges, we generate all A*-TS that can be generated by the move action. Directly after that, we check if the wait action is available at the target vertex, and then, we extend the resulting time intervals to the upper bound of the safe interval at the target vertex, where the interval is located. This is equivalent to the application of the wait actions on all A*-TS states at the target vertex. As a result, in *getSuccessors*, all valid A*-TS successors are generated (and capsulated by SIPP-IP states). So, SIPP-IP can be viewed as a modified version of A*-TS which, at every iteration, expands several states at the same time, generates all the successors of these states, and inserts them into OPEN (by definition, all generated A*-TS states are reformed into SIPP-IP states without any loss). As a result, SIPP-IP will always generate and expand all A*-TS states, and as A*-TS is complete, so is SIPP-IP. □

Lemma 1. *The A*-TS state with the minimum f-value in a SIPP-IP state $n_{SIPP-IP}(v, [t_l, t_u])$ is the state $n_{A*-TS}(v, t_l)$.*

Proof. Recall that f-value of a A*-TS state is equal to $n.t + h(n.v)$. As v of all A*-TS states in one SIPP-IP state are identical, the h-values of them are equal. As a result, the state with the minimal time i.e., t_l is the state with the minimal f-value. □

Theorem 2. *SIPP-IP is optimal.*

Proof. As SIPP-IP is a complete algorithm and the cost (time) is included as an identifier in the state, it is guaranteed that the optimal state will be expanded. Therefore, it is sufficient to prove that the first expanded state with the goal vertex is the optimal one i.e., contains the A*-TS state with optimal (minimal) time. Let us again view the SIPP-IP states as a sequence of extracted A*-TS states. Let the first expanded SIPP-IP state with the goal vertex be $n_{SIPP-IP}(v_{goal}, [t_l, t_u])$ that contains the A*-TS state $n_{A*-TS}(v_{goal}, t_l)$. Let us suppose that n_{A*-TS} is not the optimal A*-TS state, but there exists another state n'_{A*-TS} with the minimal f-value in OPEN $f(n')$ that is less than $f(n)$. According to Lemma 1, n'_{A*-TS} is located at the bottom of a SIPP-IP state $n'_{SIPP-IP}$ in OPEN, i.e., $n'_{A*-TS}.t = n'_{SIPP-IP}.t_l$. As in SIPP-IP, the states are ordered by the values $f(n_{SIPP-IP}) = n.t_l + h(n.v)$ that is equal to the f-value of the bottom A*-TS state $f(n_{A*-TS})$, the state $n'_{SIPP-IP}$ should have been expanded before the state $n_{SIPP-IP}$ because $f(n'_{A*-TS}) < f(n_{A*-TS})$, which results in a contradiction. Therefore, there is no A*-TS state with f-value less than $f(n_{A*-TS})$, and hence $n_{SIPP-IP}$ is the optimal state. □

Empirical Evaluation

We have used five different maps from the MovingAI benchmark (Sturtevant 2012) for the experiments: `empty` (sized 64x64), `room` (64x64), `warehouse` (84x170), `random` (128x128) and `Sydney` (256x256). Each map was populated with an increasing number of moving obstacles (MOs), and for each number, 200 different instances were generated that differ in trajectories of MOs. These trajectories were generated by randomly assigning start and goal locations for each MO and letting it go from start to goal using random speeds. Moreover, MO may wait at any cell for a random number of time steps. For each map, we generated instances with the densities of MOs varying from 1/25 to 1/3, where the density is the ratio of the number of MOs to the number of free cells. The start and goal locations of the agent were fixed to the top-left and bottom-right corners of the map. Note that some instances in our dataset (especially with high densities of MO) are not generally solvable.

The agent was modeled as a disk whose diameter equals the length of the grid cell. The configuration is defined as (x, y, θ, vel) where x, y are the coordinates of the cell, $\theta \in \{0°, 90°, 180°, 270°\}$ is the orientation, and $vel \in \{0, 2\}$ is the velocity of the agent.

The following motion primitives were defined: accelerating, decelerating, and uniform. Accelerating (decelerating) primitive: move with the fixed acceleration of $0.5\,\mathrm{cell/s^2}$ ($-0.5\,\mathrm{cell/s^2}$) from a configuration with zero ($2\,\mathrm{cell/s}$) velocity until reaching maximum (zero) velocity. The agent traverses four cells this way without changing its orientation. Uniform motion primitive: go with the maximum velocity ($2\,\mathrm{cell/s}$) one cell forward (the orientation does not change). Additionally, if the velocity is zero, the agent can

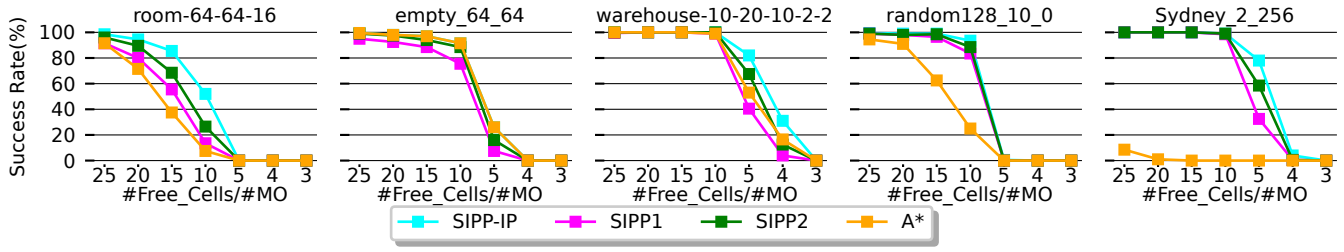

Figure 4: Success Rates of the evaluated algorithms.

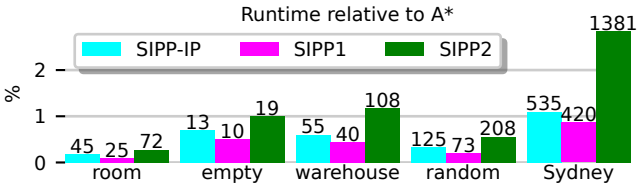

Figure 5: Runtimes of SIPP-IP, SIPP1 and SIPP2 compared to A*. Absolute values (in ms) are shown above the bars.

	room	empty	warehouse	random	Sydney	Factor
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%	

Table 1: Percentage of SIPP1/SIPP2 solutions that have higher costs compared to SIPP-IP.

rotate 90° clockwise or counterclockwise in $2\,\mathrm{s}$ or wait for one time step. The time step is chosen to be $0.1\,\mathrm{s}$.

Collision checking is conservative. We prohibit the agent and any MO from touching the same cell at the same time. Therefore, the blocked intervals of any cell were reserved for the MO when it touched this cell even partially. The agent is allowed to use a motion primitive only if it does not result in touching a cell in a blocked interval. Moreover, if the time limits of touching some cell for an agent are not integers, they are extended to the closest integers in a safer manner.

We compared SIPP-IP to A* with time steps and two straightforward extensions of SIPP: SIPP1 and SIPP2. SIPP1 is a modification that generates only kinodynamically feasible successors for the configurations where the velocity is not zero. SIPP2 further allows to re-expand the search nodes (which corresponds to allowing to reach configurations with non-zero velocities at different time steps in the same safe interval). When implementing SIPP1/SIPP2, we used the techniques from SIPPwRT (Ma et al. 2019); thus, the latter can be considered to be included in the comparison. The C++ source code of all planners is publicly available https://github.com/PathPlanning/SIPP-IP.

The experiments were conducted on a PC with Intel Core i7-10700F CPU @ 2.90GHz $\times$ 16 and 32Gb of RAM. We imposed a limit of $100,000,000$ of generated nodes for all algorithms. For each instance, we tracked whether the algorithm produced a solution and recorded the algorithm's runtime and the solution cost.

Fig. 4 presents the Success Rate (SR) plots, where SR is the ratio of the successfully solved instances to all of the instances. Indeed, all SIPP-IP competitors have lower SR (except the empty map, where A* and SIPP-IP have the same SR). For SIPP1 and SIPP2, this is explained by their incompleteness. For A* this is explained by numerous violations of the imposed limit (of $100,000,000$) on the number of the generated nodes.

For each map, we also analyzed the cost of the instances that were successfully solved by SIPP-IP, SIPP1, and SIPP2. The results are presented in Table 1. Each cell in the table tells in how many instances (in percent) the cost of the SIPP1/SIPP2 solution exceeded the cost of the optimal solution found by SIPP-IP by a fixed factor. Evidently, in most instances, the costs of competitors are larger than of SIPP-IP's costs, and for certain maps (e.g. room), the percentage of the instances where their cost notably exceeds (by more than 50%) SIPP-IP's cost is significant (up to 42%).

Finally, the algorithms' runtime analysis is presented in Fig. 5. Indeed, all versions of SIPP, including the complete one, SIPP-IP, are significantly faster than A* and reduce computation time by two orders of magnitude. Moreover, SIPP-IP is faster than SIPP2. Still, it is outperformed by SIPP1. This is expected, as SIPP1 exploits a straightforward expansion strategy missing numerous successors SIPP-IP would generate. Absolute (averaged) values of the runtime of SIPP-IP (less than $0.1\,\mathrm{s}$ on all maps, except the largest one) suggest that the proposed planner can be utilized in real robotic systems, where taking into account kinodynamic constraints may be of vital importance.

Conclusion

In this work, we have presented a provably complete and optimal variant of the prominent Safe Interval Path Planning algorithm capable to handle kinodynamic constraints. We have shown that straightforward ways to extend SIPP fail in the considered setup, and therefore a more involved algorithm is needed. The latter has been presented and analyzed theoretically and empirically. The directions for future research include embedding the suggested algorithm within the multi-agent path planning solver and conducting experiments on real robots.

Acknowledgments

This work was partially supported by the Analytical Center for the Government of the Russian Federation in accordance with the subsidy agreement (agreement identifier 000000D730321P5Q0002; grant No. 70-2021-00138).

References

Ali, Z. A.; and Yakovlev, K. 2021. Prioritized SIPP for Multi-agent Path Finding with Kinematic Constraints. In Ronzhin, A.; Rigoll, G.; and Meshcheryakov, R., eds., *Interactive Collaborative Robotics*, 1–13. Cham: Springer International Publishing. ISBN 978-3-030-87725-5.

Cohen, L.; Uras, T.; Kumar, T. S.; and Koenig, S. 2019. Optimal and Bounded-Suboptimal Multi-Agent Motion Planning. In *Proceedings of the 12th Annual Symposium on Combinatorial Search (SOCS 2019)*, 44–51.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2): 100–107.

Li, J.; Chen, Z.; Harabor, D.; Stuckey, P. J.; and Koenig, S. 2022. MAPF-LNS2: Fast Repairing for Multi-Agent Path Finding via Large Neighborhood Search. In *Proceedings of the AAAI Conference on Artificial Intelligence*.

Ma, H.; Hönig, W.; Kumar, T. K. S.; Ayanian, N.; and Koenig, S. 2019. Lifelong Path Planning with Kinematic Constraints for Multi-Agent Pickup and Delivery. In *Proceedings of the 33rd AAAI Conference on Artificial Intelligence (AAAI 2019)*, 7651–7658.

Narayanan, V.; Phillips, M.; and Likhachev, M. 2012. Anytime safe interval path planning for dynamic environments. In *Proceedings of The 2012 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2012)*, 4708–4715.

Phillips, M.; and Likhachev, M. 2011. SIPP: Safe interval path planning for dynamic environments. In *Proceedings of The 2011 IEEE International Conference on Robotics and Automation (ICRA 2011)*, 5628–5635.

Pivtoraiko, M.; and Kelly, A. 2011. Kinodynamic motion planning with state lattice motion primitives. In *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2172–2179. IEEE.

Sturtevant, N. R. 2012. Benchmarks for Grid-Based Pathfinding. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(2): 144–148.

Yakovlev, K.; and Andreychuk, A. 2021. Towards Time-Optimal Any-Angle Path Planning With Dynamic Obstacles. In *Proceedings of the 31st International Conference on Automated Planning and Scheduling (ICAPS 2021)*, 405–414.

Yakovlev, K.; Andreychuk, A.; and Stern, R. 2020. Revisiting Bounded-Suboptimal Safe Interval Path Planning. In *Proceedings of the 30th International Conference on Automated Planning and Scheduling (ICAPS 2020)*, 300–304.

Yakovlev, K.; Andreychuk, A.; and Vorobyev, V. 2019. Prioritized multi-agent path finding for differential drive robots. In *Proceedings of the 2019 European Conference on Mobile Robots (ECMR 2019)*, 1–6.