# Jump Point Search with Temporal Obstacles

**Shuli Hu**[1], **Daniel D. Harabor**[2], **Graeme Gange**[2], **Peter J. Stuckey**[2], **Nathan R. Sturtevant**[3]
[1]School of Information Science and Technology, Northeast Normal University, China
[2]Faculty of Information Technology, Monash University, Australia
[3]Department of Computing Science, Alberta Machine Intelligence Institute (Amii), University of Alberta, Canada
husl903@nenu.edu.cn,{daniel.harabor, graeme.gange, peter.stuckey}@monash.edu, nathanst@ualberta.ca

## Abstract

In 4-connected grid-based path planning one often needs to account for temporal and moving obstacles: ones that appear, disappear and which can prevent the agent from reaching its target. Such problems are common in a variety of settings (games, robotics etc.) and they can be surprisingly challenging to solve. First, because the temporal aspect increases the size of the search space; second because the search space contains many symmetric paths, indistinguishable from one another except by the order in which grid moves appear. To tackle such problems we consider a new optimal algorithm – in the style of Jump Point Search – which can identify and break these symmetries and thus improves performance; from several factors to more than one order of magnitude vs. SIPP, arguably the gold standard baseline in the area.

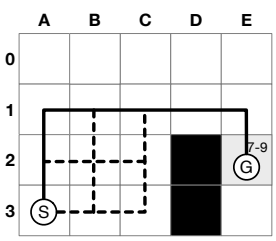


Figure 1: An optimal path from $S$ to $G$ (bold) with many spatially symmetric alternatives (dashed).

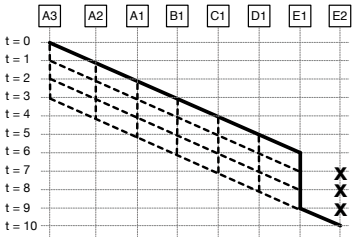


Figure 2: Time-space diagram showing an optimal path (bold) with three WAIT actions. There are many temporal symmetries (dashed).

## Introduction

We consider optimal pathfinding in 4-connected grid maps with temporal and moving obstacles. Such setups are interesting in a variety of application areas. For example, in automated warehouse logistics a mobile agent may need to avoid the trajectories of other previously planned and therefore higher priority agents. In computer games, another popular application, the map can change due to scripted or repeatedly timed events: for example a closed door is opened at a certain time, or; an agent needs to reach its target but without being crushed by moving blocks along the way.

In each of these situations the times and trajectories of every dynamic obstacle is known but problem of finding a time-optimal path is much more challenging than 2d grid pathfinding. The first issue is that the branching factor of the agent increases from 4 to 5, since waiting is now possible. The second issue is that the number of states in the environment increases, since every grid cell $(x, y)$ is time-indexed and needs to be duplicated, potentially many times. The third issue is that time only moves forward, which introduces additional complications not found in other related domains (e.g., spatial pathfinding in grids with three dimensions). The fourth issue, perhaps most significant, is symmetry: there exist in the time-expanded grid many paths which have the same start and end location, and the same cost, and which differ from one another only in the order that grid moves (UP, DOWN, LEFT, RIGHT, WAIT) appear.

Figure 1 shows an example where multiple equivalent paths exist between a start and target location. The only difference is where horizontal and vertical moves appear in the sequence of actions. We refer to such interleavings as *spatial symmetries*. Figure 2 shows a different type of equivalence which we call *temporal symmetries*. We fix the bold path of Figure 1, but assume there is a temporal obstacle at E2 at times 7-9 (shown in grey in Figure 1). In this example there exist three necessary wait actions on the optimal path from start to target. Notice that there are many equivalent solutions and each differs from all the rest only in where the wait actions are taken.

In this work we develop Temporal Jump Point Search (JPST), a new grid-based path planning algorithm that breaks spatial and temporal symmetries in time-expanded and 4-connected grid graphs. The main ingredient is a new canonical ordering (Sturtevant and Rabin 2016) called Vertical-then-Horizontal-then-Wait (VHW, for short). This ordering generates paths where vertical and horizontal moves appear as early as possible and where wait moves are taken as late as possible. We show that for any optimal time-expanded grid path there exists an equivalent path, with the same cost, which is VHW. The ordering allows JPST to prune symmetric path permutations per expanded node which can dramatically improve search performance.

JPST is closely related to Jump Point Search (Harabor

Copyright © 2021, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

and Grastien 2011; Harabor and Grastien 2014), a popular pathfinder which breaks spatial symmetries in 2d maps. Compared to that work, JPST can be understood as an extended variant which adds temporal reasoning. JPST is also closely related to Safe Interval Path Planning (SIPP) (Phillips and Likhachev 2011), another popular technique that speeds up search in time-expanded graphs. JPST breaks temporal symmetries in a similar fashion to SIPP but further enhances performance using additional spatial reasoning.

In a wide range of experiments we compare JPST to SIPP. Problems include robotics-style settings, with few temporal obstacles, and game-like settings with many thousands of temporal obstacles. Results show that JPST is often much faster, with speedups ranging from several factors to more than one order of magnitude.

## Preliminaries

A **gridmap** is a two-dimensional data structure often used in the context of pathfinding search. It consists of $w \times h$ non-overlapping square cells, each one marked as either traversable or non-traversable. Each cell is uniquely identified by its $x$ and $y$ coordinates. In this work we consider gridmaps where a cell has up to 4 adjacent neighbours, one in each cardinal direction: UP, DOWN, LEFT, RIGHT. Moving from one adjacent cell to the next is permitted and each such action has a cost of 1. We say that the move is *valid* if the origin and destination cell are both traversable.

A **time-expanded grid** is similar to a gridmap but with an added dimension: time. In this representation time elapses in episodes of equal duration known as *timesteps* and each move action advances time by exactly one timestep.

We use the notation $l@t$ to refer to the grid cell $l = (l_x, l_y)$ as it appears at timestep $t$. The neighbours of $l@t$ are time-indexed cells $l'@(t+1)$ where $l'$ is gridmap adjacent to $l$. We say that a move from $l@t$ to $l'@(t+1)$ is valid if $l@t$ and $l'@(t+1)$ are both traversable. When moving across a time-expanded grid we consider the four standard directions plus an additional move called WAIT. Waiting has a cost of 1 and its effect is to reposition the agent but only in the time dimension: i.e. from $l@t$ to $l@(t+1)$. With waits included, each $l@t$ can thus have $\leq 5$ neighbours.

A **path** $\pi = \langle l_0@t, \ldots, l_k@(t+k)\rangle$ is an ordered sequence of cells beginning with location $l_0$ at timestep $t$ and finishing at location $l_k$ at timestep $t+k$. We say that a path is *valid* if every location on the path is traversable and if there exists a valid move from each location on the path to the next. When discussing paths, we sometimes refer to their valid *continuations*. A continuation is a move $\vec{v}$ whose application extends the path by one location. We use the notation $n'@(t+k) = n@t + k \times \vec{d}$, with $k$ a positive integer, to say that node $n'@(t+k)$ is reached from $n@t$ through the application of $k$ valid moves in direction $\vec{d}$.

A **plan** is an ordered sequence of move actions $[\vec{d_1}, \ldots, \vec{d_k}]$. The actions are executed by an agent one after the other, from an initial location $l@t$. The result of a plan is to move the agent through the time-expanded gridmap and to a new location $l'@(t+k)$. We will say that each path $\pi$ is the result of executing an associated plan.

An **instance** is a pair of (non-time-indexed) grid cells $s$ and $g$ and a set of time-indexed temporal obstacles $\mathbf{O}$. An instance is *feasible* if there exists a path that avoids all permanent and temporal obstacles, beginning from $s$ at time index 0 (resp. the starting state) and finishing at $g$ at some future time index $k \geq 0$ (resp. the goal or target state).

Given an instance, our **objective** is to find an optimal path $\pi^* = \langle s@0, \ldots, g@k\rangle$ in the time-expanded grid which avoids all obstacles. The path $\pi^*$ is optimal if it minimises, among all possible paths from $s$ to $t$ the total cost of all planned actions (which for this paper is equivalent to the arrival time $k$).

## Idea

We extend Jump Point Search (Harabor and Grastien 2011) to time-expanded gridmaps with temporal obstacles. The idea is simple: we introduce a new canonical ordering called Vertical-then-Horizontal-then-Wait (VHW, for short). This ordering generates paths with vertical and horizontal moves that appear as early as possible and wait moves, which are taken as late as possible. We show that for any optimal path in a time-expanded grid there exists an equivalent path, with the same cost and arrival time, which is also VHW. We refer to this new algorithm as Temporal JPS or simply JPST.

When expanding a node, the successor set considered by JPST only includes adjacent locations where the current path can be continued in a way that is *canonical*. The canonical property is a search invariant which prevents the exploration of paths that are symmetric permutations of one another.

**Definition 1.** A path $\pi = \langle n_0, \ldots, n_k\rangle$ is *VHW-canonical* if its corresponding plan $(n_0, \sigma)$ where $\sigma = [\vec{d_1}, \ldots, \vec{d_k}]$) has: (i) the vertical-first property; (ii) the wait-last property and; (iii) the backtrack-free property.

- *Vertical First*: there exists in $\sigma$ no subsequent pair of actions, with $\vec{d_i}$ a horizontal move and $\vec{d_{i+1}}$ a vertical move, which could be swapped to generate a new valid path $\pi'$ from $n_0$ to $n_k$.
- *Wait Last*: there exists in $\sigma$ no subsequent pair of actions, with $\vec{d_i}$ a wait move and $\vec{d_{i+1}}$ a non-wait move, which could be swapped to generate a new valid path $\pi'$ from $n_0$ to $n_k$.
- *Backtrack Free*: there exists in $\sigma$ no subsequent pair of actions, with $\vec{d_i} \neq \vec{d_{i+1}}$ which could be replaced with a pair of subsequent WAIT actions, to generate a new valid path $\pi'$ from $n_0$ to $n_k$.

Figure 1 shows an example (bold) where the Vertical-First property breaks spatial symmetries. The property guarantees that when JPST generates a node, the path to that node will feature vertical moves as early as possible. Figure 2 meanwhile shows how Wait-Last breaks temporal symmetries. Notice the optimal path (bold) features wait moves only after all possible vertical and horizontal moves.

## Pruning Rules

Each time JPST expands a node we apply a neighbour pruning rule to reduce the set of adjacent cells. Pruning

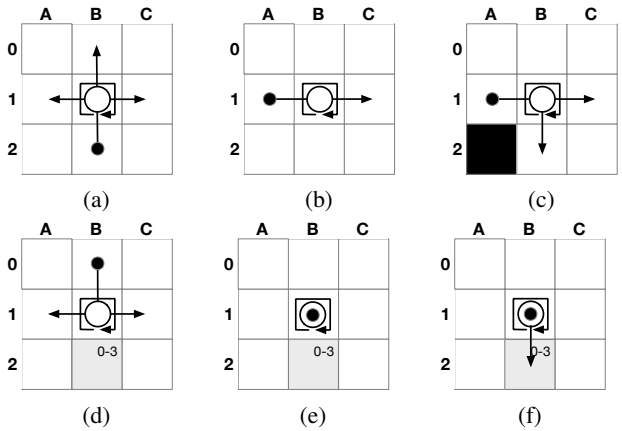


Figure 3: JPST moves to $B1$ in a variety of different situations. Some tiles (filled black) are blocked at all timesteps. Others (filled grey) are labelled with the timesteps during which they are blocked. For (d)-(f) B2 has a temporal obstacle at time 0-3. The parent is indicated by a dark dot and arrows indicate neighbours which are VHW canonical. Other neighbours are non-canonical and can be pruned.

eliminates neighbours whose continuation fails to preserve the VHW ordering. Figure 3 shows a number of examples. We discuss each example in turn.

**Spatial pruning:** In Figure 3(a) JPST moves UP. Of the five possible continuations, only DOWN is pruned by the VHW ordering (this node violates the backtrack-free property; i.e. we could have simply waited to reach the same location with the same cost). By comparison, in Figure 3(b) JPST moves RIGHT. Here the only canonical continuations are RIGHT and WAIT. Notice that LEFT does not preserve backtrack-free while UP and DOWN are not vertical-first (i.e., there exists a path to each of $B0$ and $B2$ where the vertical moves appear sooner). In Figure 3(c) we show another situation where JPST passes next to an obstacle. Here $B2$ cannot be pruned as the equivalent path where vertical moves appear sooner is blocked. In such cases we say that $B2$ is *forced* to be generated.

**Temporal pruning:** Figures 3(d)-3(f) show a running example over several timesteps. In Figure 3(d) JPST moves DOWN but a temporary obstacle impedes further progress. In Figure 3(e) JPST waits for the obstacle to clear. Notice that after a wait we can usually prune all neighbours save one. In Figure 3(f) JPST is again *forced* to generate $B2$, and it arrives there just as the obstacle disappears.

**Theorem 1.** Every optimal path $\pi = \langle l_0@0, \ldots, l_k@k \rangle$ can be transformed into an equivalent path $\pi'$ which is VHW-canonical by reordering the moves in its corresponding plan.

*Proof.* (Sketch) There are three categories of actions: Horizontal (H), Vertical (V) and Wait (W). These can be combined in only six different ways to produce a canonical turning point where an optimal path can bend. These are: WH, WV, HW, HV, VW, VH. Consider any turning point along the optimal path $\pi$. Either the turning point follows the VHW-canonical ordering, or we can re-arrange the two moves that make up the turning point such that vertical actions appear sooner and wait actions appear later (as appropriate for the situation). We continue to re-write turning points in this way until no further modifications are possible. Notice that the path produced by this revised plan is valid and has the same cost as the original. This new path is therefore optimal and is also VHW-canonical. □

**Algorithm 1** The function `get-successors`

**Require:** $x@t$: current node, $\vec{d}$: direction, $g$: goal
1: $S \leftarrow \emptyset$
2: $M \leftarrow$ get-canonical-moves$(x@t, \vec{d})$
3: **for all** $\vec{d'} \in M$ **do**
4: $\quad S \leftarrow S \cup jump(x@t + \vec{d'}, \vec{d'}, g)$
5: **return** $S$

**Algorithm 2** The function `jump`

**Require:** $x@t$: current node, $\vec{d_1}$: direction, $g$: goal
1: $M \leftarrow$ get-canonical-moves$(x@t, \vec{d_1})$
2: **if** $x == g$ **or** $\exists\ \vec{d_2} \in M \mid \vec{d_2}$ is forced **then**
3: $\quad$ **return** $\{(x@t, \vec{d_1})\}$
4: $S \leftarrow \emptyset$
5: **if** $t <$ max time index of any disappearing obstacle **then**
6: $\quad x'@(t+1) \leftarrow x@t + \mathit{wait}$
7: $\quad S \leftarrow S \cup jump(x'@(t+1), \mathit{wait}, g)$
8: **for all** $\vec{d_2} \in M$ such that $\vec{d_2}$ is horizontal **do**
9: $\quad x'@(t+1) \leftarrow x@t + \vec{d_2}$
10: $\quad S \leftarrow S \cup jump(x'@(t+1), \vec{d_2}, g)$
11: **for all** $\vec{d_2} \in M$ such that $\vec{d_2}$ is vertical **do**
12: $\quad x'@(t+1) \leftarrow x@t + \vec{d_2}$
13: $\quad S \leftarrow S \cup jump(x'@(t+1), \vec{d_2}, g)$
14: **return** $S$

## Jumping Rules

One way pruning rules can improve the performance of search is in the context of the current node where they can reduce the number of generated successors per expansion step. Another way pruning rules can help is in the context of the candidate successor set; i.e. for nodes where we have already established there exists a VHW-canonical continuation of the path. When applied to such candidate successors, pruning rules can sometimes reduce the branching factor to 0 or 1. Rather than adding such nodes to the A* OPEN list we propose to immediately expand and apply the rules anew. The recursion continues until the next move is invalid (in which case we say that the branch is a dead-end) or until the recursion produces, and returns as a successor, a type of node which we call a *temporal jump point*.

**Definition 2.** Let $l@t$ be a time-indexed location in the grid which is generated as a result of applying move $\vec{d_1}$ at location $p$ during timestep $(t-1)$; i.e. $p@(t-1) + \vec{d_1} = l@t$. We say that the tuple $(l@t, \vec{d_1})$ is a *temporal jump point* if: (i) $l@t$ is the target or; (ii) if $l@t$ has a valid move $\vec{d_2} \neq \vec{d_1}$ such that:

- the plan $p@(t-1) + \vec{d_1} + \vec{d_2}$ is valid;

- the plan $p@(t-1) + \vec{d_2} + \vec{d_1}$ is invalid.
- $\vec{d_2}$ comes before $\vec{d_1}$ in the VHW ordering.

The definition says a node $l@t$ is jump point if that node is the target or if $l@t$ has a neighbour which is *forced*; i.e. it can only be locally reached by going against the VHW ordering, since the preferred plan $p@(t-1) + \vec{d_2} + \vec{d_1}$ is invalid.

During recursion we process canonical continuations in a specific partial order: first we consider nodes reached with a WAIT action; then we consider the horizontal actions, RIGHT and LEFT; finally we consider the vertical actions, UP and DOWN. This ordering guarantees we only ever recurse along paths which are VHW-canonical. That means before every horizontal move we explore whether the path can bend around an obstacle via a sequence of wait actions. Likewise, before every vertical move, we explore whether the path can bend via a sequence of horizontal actions.

**Example 1.** Consider the instance in Figure 1 with its corresponding temporal dimension shown in Figure 2. JPST expands the start state $A3@0$. This node has no parent so every valid move forms a VHW-canonical continuation of the path. There are three moves: UP, RIGHT and WAIT.

We first apply our pruning rules to the node $A3@0+$WAIT and we reduce its branching factor to 1 (again, WAIT). Notice however that no matter how much further we might recurse, there is never any cell adjacent to $A3$ which becomes unblocked at any future time. In other words this branch will never be forced to change direction and it can never produce the target. JPST will prune this branch as a dead-end once the recursion reaches the maximum timestep associated with any temporal obstacle.

The node $A3@0 +$ RIGHT is processed next but this too proves to be a dead-end. In particular, after each step RIGHT the path can WAIT but every such continuation is a dead-end. After two steps RIGHT obstacles impede further progress.

The node $A3@0 +$ UP is more fruitful. After two steps (i.e. from $A1@2$) the RIGHT continuation yields the jump point $(E1@6, \text{RIGHT})$. We add this node to the list of successors for $A3@0$ and we continue recursing: from $A1@2$ in the direction UP. Again we encounter a dead-end. Now $(E1@6, \text{RIGHT})$ is the top node the OPEN list and the only possible action is WAIT; the search having considered, and pruned, all other equivalent ways to reach the target. Recursing in the WAIT direction generates the jump point $(E1@9, \text{WAIT})$ which has just one continuation: DOWN. Expanding $(E1@9, \text{WAIT})$ produces the target.

Notice how recursive pruning allows the search to *jump*, from one location to the next, all without adding to and removing from the A* OPEN list any intermediate nodes along the way. Our approach can be implemented as a modification of the usual `get-successors` function of grid A*, as per Algorithm 1, but where successor nodes are generated by a recursive jumping procedure. We give a pseudo-code description of this function in Algorithm 2. Line 1 identifies the set of canonical continuations for the node $x@t$. The vector $\vec{d_1}$ denotes the last move on the canonical path, from $s@0$ to $x@t$. Lines 2-3 immediately add $x@t$ as a jump point successor if this node is the target or if it has any forced continuations. The remainder of the function (Lines 5-14) simply applies recursive pruning to each non-forced canonical continuation. Notice (Line 5) that we never explore WAIT branches after the time of the last disappearing obstacle. Once this timestep has passed either there exists a spatial path to the target, or else the target cannot be reached. No amount of further waiting can help.

**Theorem 2.** JPST is optimal

*Proof.* (Sketch) We proceed by induction over the expansion of states on an optimal canonical path, one state of which is always on OPEN until the target is expanded. This induction is straightforward, so we focus on showing that the entire path can be generated through successive expansions.

Let $\pi$ be an optimal VHW-canonical path. Now divide $\pi$ into segments such that each segment is produced by taking as many moves as possible in a single direction. Notice that the beginning and end of every segment, except the start and target, is a turning point. Recall that every turning point on a VHW-canonical path is a temporal jump point. We apply Algorithm 2 to jump from the node at the beginning of each segment to the node at the end of the segment, without stopping at any point in between. Only the start and target remain unaccounted for. In Definition 2, the target is always returned as a temporal jump point and thus also by Algorithm 2. For the start, recall that JPST expands this node and considers every valid move and each of its corresponding canonical continuations. □

## Further Examples

We now discuss several further examples of searching with JPST. We demonstrate why optimal time-expanded search is more challenging than conventional grid search and why conventional algorithms such as JPS are insufficient.

**Example 2.** In Figure 4(b) we show a time-expanded version of the map in Figure 4(a). The start and target are adjacent but the direct path is blocked and the start soon will be. From $S$, JPST has two canonical continuations: WAIT and DOWN. The first is proven to be a dead-end in 2 steps. The second yields a single successor, the temporal jump point $A2@2$ which has a forced successor $A3@3$. Expanding $A2@2$ produces the jump point $A3@5$ with $A2@6$ being forced, and $A5@8$ with $A4@9$ being forced. Expanding $A3@5$ allows JPST to reverse $y$-direction and in 3 steps it produces the target, $A0@8$. Notice how we systematically explore the entire search space but use only canonical paths: waiting before each horizontal step and continuing again as soon as adjacent obstacles disappear. Notice also that the optimal path contains spatial cycles.

**Example 3.** In Figure 4(c) JPST solves a small problem in three dimensions. Notice locations $C0 - C3$ are blocked at timesteps 0–7. When JPST recurses along row 3, it generates one jump point adjacent to each of these locations, since at timestep 7 – after a sequence of wait actions – there exists in each case a forced neighbour. The example is interesting because only one of these jump points, $D0@6$ is not a dead-end. The others are generated but the branches are pruned at timestep 8 after a forced move LEFT.

The examples, especially Figures 4(c) and 4(d), show that more aggressive symmetry breaking (in the form of speculative recursions) can eliminate unpromising branches and further speed up search – not just for JPST but also JPS. A more detailed discussion is a topic for further work.

## Practical Considerations

We now discuss a number of additional considerations which arise for JPST in practice.

**Concrete paths:** JPST returns a path of temporal jump points but not necessarily a concrete path that can be readily executed. However from any path produced by JPST it is easy to derive a complete canonical path as follows. Simply take the delta between every two adjacent jump points on the JPST path: in the $x$-, $y$- and $t$-dimension. We then generate a VHW-canonical subpath with $\delta y$ vertical steps followed by $\delta x$ horizontal steps and $\delta t$ wait steps. In other words, the sequence of $\geq 1$ canonical moves between two temporal jump points follows the pattern V*H*W*. Moreover, every such subpath is guaranteed to be valid as well as optimal.

**Jump Limits:** JPST can perform a great deal of grid scanning. The scans are useful if they produce must-expand successors, otherwise they can be regarded as wasted work. To prevent JPST from scanning large and uninteresting areas we apply a jump limit $jl$ that stops each scan after a fixed number of steps. When the limit is reached the node currently scanned is added to the OPEN list as a pseudo jump point successor. Similar ideas appear in bounded JPS (Sturtevant and Rabin 2016) and for the same reason. In experiments we use $jl = 256$ for map `gardenofwar` and `Sirocco`, and $jl = 128$ for all other maps.

**Fast Scanning:** One critical aspect for JPST performance is that grid scanning operations proceed efficiently. In (Harabor and Grastien 2014) authors propose *block jumping*, a speedup technique for JPS which improves performance by increasing the size of the stride during scanning operations. We consider a similar improvement for JPST.

To speed up horizontal scanning we use a bitfield representation of the map, stored in row major order, to indicate the position of permanent obstacles. Another similar bitfield indicates the position of temporal obstacles. The only possible locations of jump points are at cells adjacent to the locations indicated by the bitfields. Using only simple bit scanning and bit manipulation operators we then construct a forced neighbour checking procedure that allows JPST to recurse horizontally across the grid in steps of size 32, each time reasoning about all adjacent nodes and their neighbours, simultaneously. Since the adaptation of this idea is straightforward we omit the mechanical details. A full description is available in (Harabor and Grastien 2014).

**Speculative Waiting:** JPST generates a WAIT successor each time it scans a grid node where temporal obstacles appear and disappear. In Figure 4(d), JPST expands $A4@0$ and scans RIGHT to reach $B4@1$. The search may eventually generate up to three WAIT successors: $B4@3$, $B4@5$ and $B4@77$. Such WAIT successors are necessary to maintain the VHW-canonical property but adding them to the OPEN list can introduce unnecessary overheads: in this case, none of the three nodes can appear on the optimal path.

To mitigate such issues we propose to immediately expand all reachable wait successors and generate in their stead any corresponding forced neighbours. In the example, this strategy allows us to immediately prove each WAIT successor leads to a dead-end. We reduce the size of the OPEN list (nothing is added) and the search makes faster progress toward the target. Where a dead-end proof is unavailable, speculative waiting can still help: by generating forced neighbours instead of WAIT successors we grow the $f$-value faster and again potentially speed up search.

**Safe Intervals:** In our discussion thus far we have assumed the search space is a completely specified time-expanded grid graph. But such a representation may be prohibitive under a long planning horizon $T$: there are $w \times h \times T$ grid cells that need to be stored and potentially scanned as JPST moves through the environment.

We can speed up temporal symmetry breaking by keeping for each $(x, y)$ location a list of *intervals* that specify when that location is traversable. Equivalently, we store a list of the times when $(x, y)$ is a temporal obstacle. Recall that this data is given as part of the instance specification. Suppose now that we need to decide if location $n@t$ has a forced continuation after some $k \geq 0$ number of WAIT actions.

- We scan the list of traversable intervals for $n = (n_x, n_y)$ and find the one where $t \in [t_b, t_e]$.
- We scan the list of intervals stored with each adjacent cell $n' = (n'_x, n'_y)$. and we generate any which overlaps with $[t_b, t'_e]$; i.e., JPST waits for $k \leq t_e - t$ timesteps and then moves in a corresponding forced direction, arriving in the adjacent safe interval as early as possible.
- Each safe interval has an associated $g$-value. This value can be improved in the usual way: by finding a shorter path which reaches the interval earlier. The $g$-value is *settled* once the interval is expanded and any subsequent paths through such an expanded interval can be pruned.

Safe Intervals and Speculative Waiting are closely related to *Safe Interval Path Planning* (SIPP) (Phillips and Likhachev 2011). Like SIPP, we reason about time in terms of intervals but we break temporal symmetries by taking WAIT actions as late as possible (SIPP does not commit to any concrete temporal plan). We also interleave temporal symmetry breaking with spatial symmetry breaking which allows us to generate and expand fewer nodes than SIPP.

**Obstacles with Trajectories:** The domain model of JPST is a time-expanded grid where obstacles have fixed spatial coordinates but can appear and disappear at different times. In other settings, notably Multi-Agent Path Finding (MAPF), obstacles represent other agents whose trajectories we must avoid to prevent collisions. There are two types of MAPF collisions: vertex collisions, where two agents attempt to occupy the same time-indexed location at the same time, and edge collisions, where two agents attempt to swap $(x, y)$ positions.

JPST jump rules account for vertex collisions but not edge collisions. Consider Figure 4(d) where agent $A1$ attempts to move to $A2@2$ and $A2$ attempts to move to $A3@2$. The origin and destination vertices of both agents is traversable

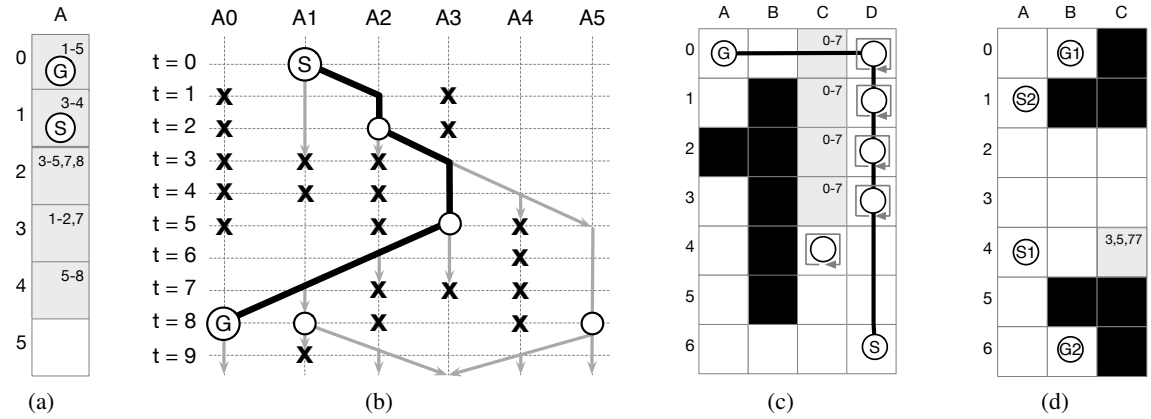


Figure 4: **(a).** A simple one dimensional map. **(b).** JPST searches for a path from $S$ to $G$ of the map of Figure 4(a). We show the optimal solution (solid black), other canonical continuations (solid grey) and all temporal jump points (circles). **(c).** JPST searches for a path from $S$ to $G$. We show the optimal solution (bold lines) and the $(x, y)$ position of temporal jump points identified as successors of the start node. **(d).** We show two agents at $S1$ and $S2$. If each follows the unique optimal path to their respective target locations $G1$ and $G2$ there is an edge collision at timestep 2 on the edge (A2,A3).

at the beginning and end of the action and our pruning rules do not generate any forced neighbours for either agent.

We can adapt JPST to handle such cases by taking an edge-centric view of the time-expanded domain. That means reasoning explicitly about outgoing edges of each time-indexed location and enabling or disabling moves depending on the situation. One way to achieve this involves storing with each safe interval a list of constraints on outgoing edges for the timesteps bounded by the interval. Our definition of forced neighbours is likewise easily adapted: we check not only the nodes on equivalent path are traversable but also that the necessary edges are enabled.

## Evaluation

We evaluate JPST vs. SIPP (Phillips and Likhachev 2011) in two distinct settings: Pathfinding with Moving Obstacles and Pathfinding with Constraints. Our maps and instances are drawn from Sturtevant's well known benchmark set (Sturtevant 2012). Both SIPP and JPST are implemented in C++ and both are based on code from `libMultiRobotPlanning`, a freely available pathfinding library due to Wolfgang Hönig.[1] Our implementation is also available. [2]

### Experiment #1: Moving Obstacles

In this experiment we consider pathfinding among groups of moving obstacles with known/predictable trajectories. Setups such as these can be found in crowd simulation scenarios, collaborative robot applications (e.g. Amazon Canvas) and in computer video games.

To generate instances we select four maps drawn from real games: `gardenofwar`, `orz700d`, `Sirocco` and `w_woundedcoast`. Each has an associated *scenario file* that specifies several thousand single-agent pathfinding instances (i.e. start-target location pairs). We solve each instance in turn and treat optimal plans from the previous $k$ instances as moving obstacle trajectories, to be avoided when solving instance $k + 1$. For the first $k$ problems we generate obstacle trajectories by solving the *last* $k$ problems in the scenario file. The obstacle trajectories are just the optimal plans returned by SIPP. That means both JPST and SIPP solve the same problem instances and both must avoid the same set of obstacle trajectories.

Results appear in Figure 5 using cactus plots. We sort the instances by ratio of time for SIPP over JPST for each $k$ value. This shows the full distribution of behaviour. When the ratio $> 1$ then JPST is better. We use both a Manhattan distance heuristic, and a perfect 2D heuristic (the exact distance to the target ignoring temporal obstacles), which we compute offline. The $k = 0$ case shows the maximum advantage that spatial symmetry breaking can give.

We observe that JPST improves on SIPP in a majority of cases, with speedups from several factors to over one order of magnitude. The speedups for JPST are larger when the two algorithms rely on a Manhattan heuristic, and on larger maps, such as `gardenofwar` and `Sirocco`, where open space introduces more spatial symmetries. In corridor-like maps, such as `orz700d` and `w_woundedcoast`, spatial symmetries are fewer and the advantage for JPST is reduced. As $k$ increases, the gap narrows: there are more locations with temporal obstacles which reduces jump distances and forces JPST to expand more nodes. For $k = 100$ the performance of JPST is similar to SIPP. Both methods break temporal symmetries but spatial symmetries are few. Here JPST can incur some small overheads due to grid scanning. Adding still more obstacles, in an adversarial manner, can eliminate even temporal symmetries. In this case both algorithms will converge to time-expanded A*.

[1] https://github.com/whoenig/libMultiRobotPlanning

[2] https://github.com/husl903/Multi-pathfinding

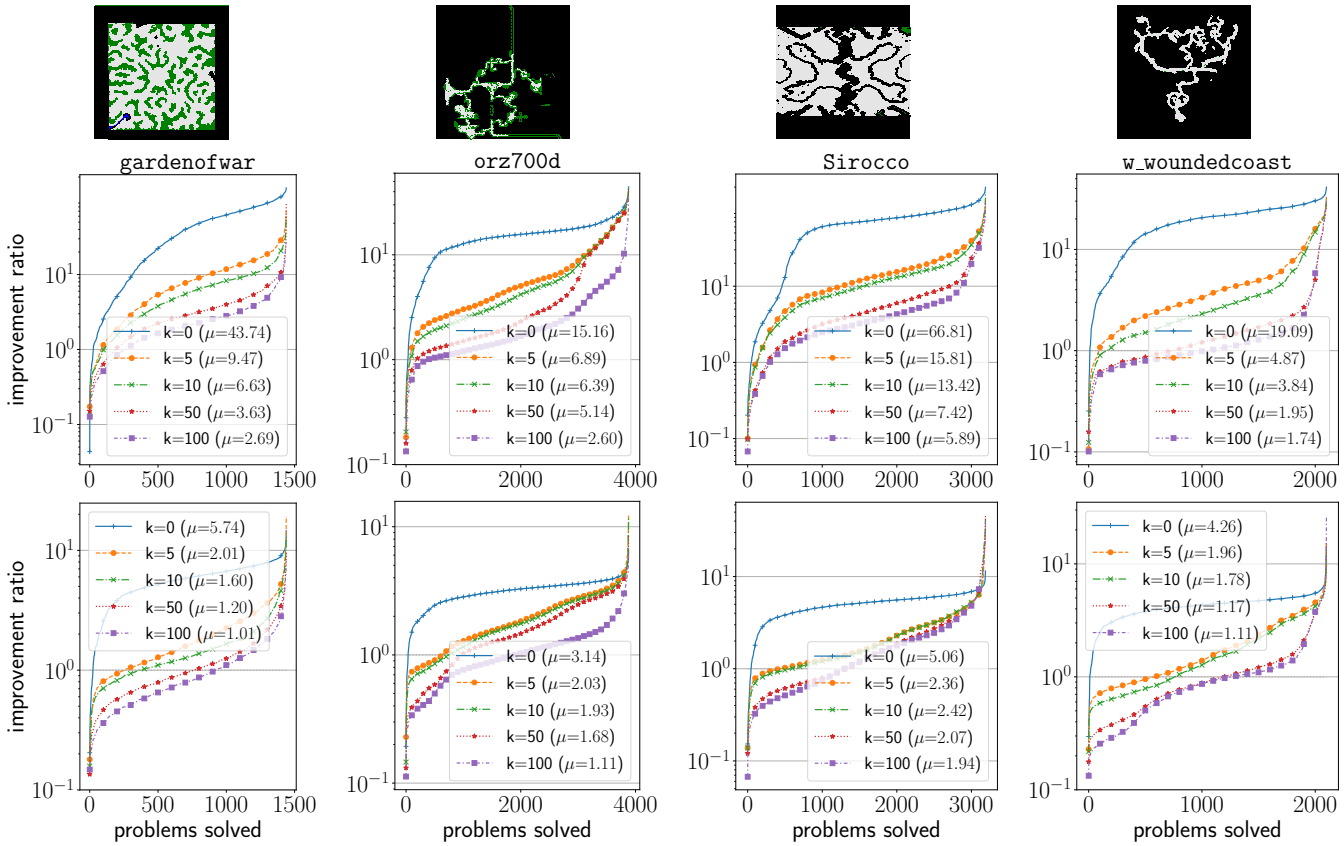


Figure 5: Moving obstacles experiment. We insert $k \in \{0, 5, 10, 50, 100\}$ moving obstacle trajectories on the map and compute optimal paths with SIPP and JPST. In the top row, both algorithms use a Manhattan estimator. In the bottom row, both algorithms use a precomputed Perfect 2D estimator. We report runtime ratios (SIPP time over JPST time). The dotted line is a ratio of 1.0. Higher is better with values $> 1.0$ indicating performance-improving results for JPST. Symbol $\mu$ indicates average ratio.

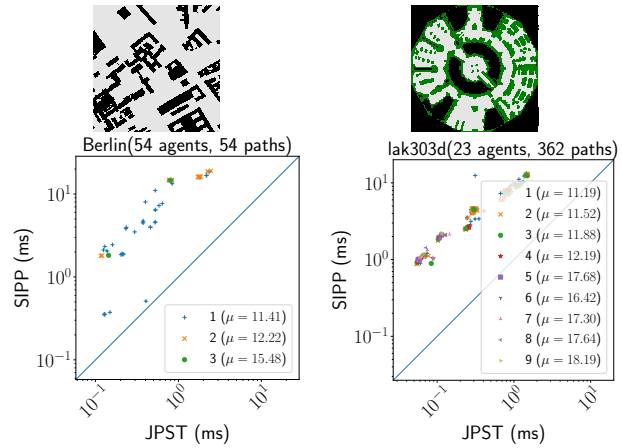


Figure 6: Runtime scatter plots for SIPP vs. JPST on CBS-derived problem instances. Both algorithms use a Manhattan heuristic. Shapes denote number of temporal obstacles (max 9). Symbol $\mu$ indicates average time ratio (SIPP/JPST).

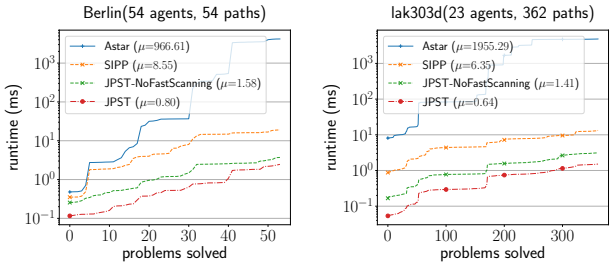


Figure 7: Cactus plot for CBS-derived problem instances. We compare A*, SIPP and JPST (with and without fast scanning). All algorithms use a Manhattan heuristic. Symbol $\mu$ indicates average performance.

## Experiment #2: Pathfinding with Constraints

In some applications (e.g. autonomous warehouse logistics) agents plan in time-expanded grids but subject to some small set of operating constraints (cf. 2K to 20K temporal obstacles to be avoided, as in the previous experiment).

To generate operating constraints we solve Multi-agent Path Finding (MAPF) problems using Conflict-based Search

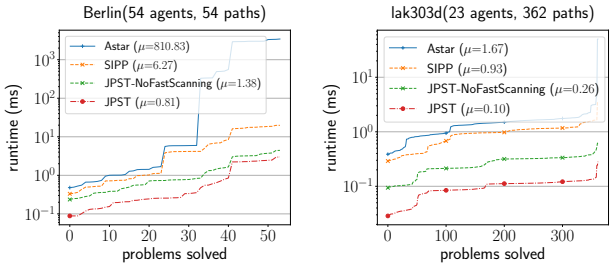


Figure 8: Cactus plot for CBS-derived problem instances. We compare A*, SIPP and JPST (with and without fast scanning). All algorithms use a Perfect 2D heuristic. Symbol $\mu$ indicates average performance.

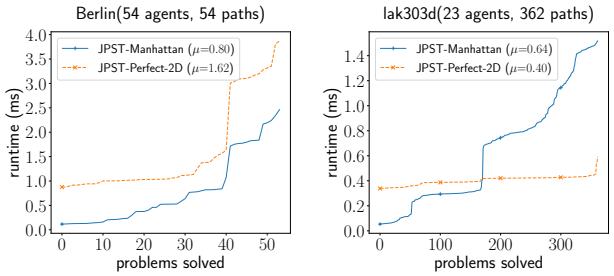


Figure 9: Cactus plot comparison between JPST with an online-computed perfect 2D heuristic and JPST with a JPST with Manhattan distance. We show performance on all CBS-derived problems. Symbol $\mu$ indicates average run time.

(CBS) (Sharon et al. 2015). We use Hönig's CBS implementation on two known benchmark maps: `Berlin` and `lak303d`. For each map we take the CT tree of the largest MAPF problem that CBS could solve with a 5 minute timeout. We collect all the path planning problems solved at each node in the CT tree and the corresponding constraints. We then re-solve each of these problems, this time with SIPP and JPST. We run both methods with two distinct heuristics (Manhattan and Perfect 2D) and give separate comparisons.

Comparative results for JPST and SIPP, using Manhattan heuristic are shown in Figure 6 and 7. Here the average improvement ratio for JPST is 11.7 on the map `Berlin`, and 12.0 on `lak303d`. Meanwhile Figure 8 shows comparative results with a perfect 2D heuristic. Here we measure raw runtimes. Again JPST has a clear advantage with reductions in average running time of up to several factors vs SIPP and up to several orders vs Time Expanded A*, which is the default path planner in our implementation of CBS. We also illustrate the significant benefits of fast scanning by comparing against JPST with this optimisation turned off.

In the context of CBS it is usually worthwhile to compute the perfect heuristic for each agent, since one typically replans the path of every agent many times. If the target changes frequently however the cost of computing the heuristic, plus the faster running time, can be a net loss vs. searching with a less informed heuristic. In Figure 9 we compare JPST running time when the cost of the perfect 2D heuristic is included in the total time required to solve each instance. For the `Berlin` instances Manhattan is usually faster while for the `lak303d` instances the results are mixed. Notice however that the absolute difference is just milliseconds. These results indicate the main performance improving benefits of JPST-Perfect-2D come from strong symmetry breaking rather than a strong estimator.

## Conclusion

We introduce Temporal Jump Point Search (JPST), a new algorithm for pathfinding in 4-connected maps with temporal obstacles. Fast, optimal and entirely online, JPST effectively breaks both spatial and temporal symmetries using a new canonical ordering scheme called Vertical-then-Horizontal-then-Wait (VHW). JPST is closely related to SIPP, a leading planner from the literature which only breaks temporal symmetries. We compare these two algorithms in a range of experiments, with few temporal constraints and with up to tens of thousands. Results show that in both cases JPST can be substantially faster, with speedups from a few factors to over one order of magnitude. Further work in this area could consider other types of grid symmetries. For example, 8-connected grids are a popular domain model in 2d (spatial) pathfinding but the addition of time in this case makes symmetry breaking substantially more complicated since cost and time no longer coincide.

## Acknowledgements

This research was supported by NSFC #61976050 to Shuli Hu. This work was also funded by the Canada CIFAR AI Chairs Program. We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC). Research at Monash University was partially supported by the Australian Research Council (ARC) under grant numbers DP190100013 and DP200100025 as well as a gift from Amazon.

## References

Harabor, D. D.; and Grastien, A. 2011. Online Graph Pruning for Pathfinding on Grid Maps. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 1114–1119.

Harabor, D. D.; and Grastien, A. 2014. Improving Jump Point Search. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, 128–135. URL http://www.aaai.org/ocs/index.php/ICAPS/ICAPS14/paper/view/7914.

Phillips, M.; and Likhachev, M. 2011. SIPP: Safe Interval Path Planning for Dynamic Environments. In *2011 IEEE International Conference on Robotics and Automation*, 5628–5635. IEEE.

Sharon, G.; Stern, R.; Felner, A.; and Sturtevant, N. R. 2015. Conflict-based Search for Optimal Multi-agent Pathfinding. *Artificial Intelligence* 219(Supplement C): 40 – 66. ISSN 0004-3702. URL http://www.sciencedirect.com/science/article/pii/S0004370214001386.

Sturtevant, N. R. 2012. Benchmarks for Grid-Based Pathfinding. *Transactions on Computational Intelligence and AI in Games* 4(2): 144 – 148. URL http://web.cs.du.edu/~sturtevant/papers/benchmarks.pdf.

Sturtevant, N. R.; and Rabin, S. 2016. Canonical Orderings on Grids. In *International Joint Conference on Artificial Intelligence*, 683–689. New York, USA.