X*: Anytime Multi-Agent Path Finding For Sparse Domains Using Iterative Repairs

Kyle Vedder, Joydeep Biswas

University of Massachusetts Amherst, College of Information and Computer Sciences
140 Governors Drive, Amherst, MA 01002, United States of America

Abstract

Optimally solving the Multi-Agent Path Finding (MAPF) problem is computationally expensive; often, real world systems need a *valid* solution quickly and would like to improve the quality of the solution if time allows, making anytime MAPF solvers of great practical interest. In many of these practical domains, planning for each agent individually produces a solution with *sparse* agent-agent collisions involving few agents, and these collisions allow for a geometrically local repair, even if this repair is globally sub-optimal. Furthermore, if the geometric scope of the repair is increased, it will produce a higher quality repair; as the scope grows to the entire domain, it will produce a globally optimal repair. Using these properties, we present an anytime MAPF framework, WAMPF, along with an efficient implementation, X*, which employs re-use techniques during repair scope growth. On the theoretical side, we provide full proofs of correctness for WAMPF and X*. On the experimental side, we show that X* outperforms state-of-the-art anytime or optimal MAPF solvers in fast solution generation and is competitive with state-of-the-art anytime or optimal MAPF solvers for optimal solution generation.

Keywords: Multiagent Systems, Motion and Path Planning, Multi-Agent Path Finding, Anytime Path Finding

1. Introduction

Constructing a minimal cost, collision free path from a known start state to a known goal state for a single agent in the face of obstacles and under time constraints is a problem faced in many domains, from robotics to videogame agents. This problem, known as the Single-Agent Path Finding problem (SAPF), appears in domains with both discrete and continuous state spaces.

In discrete spaces, the problem can be modeled in a variety of ways, including integer linear programming [1, 2], satisfiability [3], and answer set program-

Email addresses: kyle.c.vedder@gmail.com (Kyle Vedder), joydeepb@cs.umass.edu (Joydeep Biswas)

ming [4]; however, solutions most commonly model the problem as a graph with vertices that represent a state in the state space and with edges that represent the valid transitions between these states. Combinatorial search algorithms such as A* [5] are then used to find minimal cost paths between the start vertex and the goal vertex on the graph, and the resulting path can be mapped directly to a minimal cost set of transitions from the start state to the goal state.

In continuous spaces, the most computationally challenging problems are intractable; for linked polyhedra moving through three-dimensional space with a fixed set of polyhedral obstacles, commonly known as the Moving Sofa problem or the Couch Mover's problem, finding an optimal, collision free path is PSPACE hard [6]. A common way to simplify continuous problems is to convert them to discrete problems; this is often done by imposing a grid-structure, such as a four-connected grid or an eight-connected grid, or by randomly sampling the space. Imposing a grid adds additional structure to the problem that can be exploited to speed search [7], but environments can be adversarially designed to admit no collision free path along a given grid, but admit many collision free paths in the continuous space version of the problem. To address this problem, the search space can be sampled online, ensuring probabilistic completeness [8]; two common ways this can be done is by constructing a random graph and then searching it [9] or in lock-step with search by constructing the data structure during search [10].

However, the problem of finding collision-free paths for *multiple* agents that also avoid colliding with each other, known as the Multi-Agent Path Finding problem (MAPF), presents another layer of difficulty. Not only is the continuous, two dimensional case of path finding for multiple rectangles, a simplification of the Couch Mover's problem setup, PSPACE hard [11], the discrete MAPF problem is also significantly more challenging than the discrete SAPF. In general, planning jointly for all agents requires planning in a state space with the dimensionality that is at least linear in the number of agents, meaning the cardinality of the state space is at least exponential in the number of agents. Under common conditions, SAPF operates on a *polynomial* domain, i.e. the difficulty of the problem grows polynomially relative to the depth of the optimal solution due to duplicate detection; under these same conditions, MAPF operates on an *exponential* domain, i.e. the difficulty of the problem grows exponentially in the depth of the solution [12, 13]. Similar to the SAPF, discrete MAPF problems can be modeled via integer linear programming [14], satisfiability [15, 16, 17], and answer set programming [18], but many solutions operate directly on graphs [19, 20, 21].

Unfortunately, in addition to being a difficult problem, the MAPF problem is also a pressing one; many multiagent systems, from warehouse automation systems to RoboCup teams, require plans for their agents that move each agent from their current state to their desired goal, without collisions, in an optimal or near-optimal manner.

In this work we focus on solving the MAPF problem in discrete domains that have *sparse* agent-agent collisions when each agent plans individually. Informally, sparse domains have few agents involved in any single agent-agent collision, and these collisions are sufficiently far apart that the problem can be reasonably decomposed.

We present seven contributions in this paper. 1) we present a novel formalism of several properties that A*'s data structures uphold, along with proofs. 2) we present an *anytime* path finding framework that exploits domain sparsity to quickly repair each collision locally, allowing for fast first solution generation 3) we present a proof of correctness and convergence to optimality for this framework. 4) we present a naïve implementation of this framework. 5) we present an efficient implementation of this framework that reuses information between searches to speed successive solution generation. 6) we present a proof of correctness of this efficient implementation. 7) we present experimental results and analysis comparing our work to the state-of-the-art which demonstrate a significant performance advantage over state-of-the-art anytime solvers for time to first solution while still providing competitive time to optimal solution compared to state-of-the-art optimal solvers. An earlier version of this work presented a similar version of the anytime path finding framework, the naïve framework implementation, and the efficient framework implementation along with proof sketches [22], but this work provides more detail, full proofs with appropriate formalisms, and a completely new experimental results section.

2. Related Work

To put our contribution in the context of the state-of-the-art, we review existing approaches related to Bounded Search (Section 2.1), Search Reuse (Section 2.2), Anytime Path Planning (Section 2.3), Multiagent Path Planning (Section 2.4), and Anytime Multiagent Path Planning (Section 2.5).

2.1. Bounded Search

Bounded Search is a technique where artificial limits are placed on the search space. While bounds usually produce a suboptimal solution, they prevent planning far into the future on a model of the world that is less likely to be accurate, thereby speeding solution generation. This bound can be enforced via the time domain such as with a time-bounded lattice [23], via depth of search such as Hierarchical Cooperative A* [21], or via restricted cost propagation such as Truncated D* Lite [24].

2.2. Search Reuse

Search Reuse is a technique where information from one or more previous searches is used to speed up future searches. One of the most famous of such algorithms, D* Lite [25], propagates changes in the environment back up the search tree, only modifying states g-values as needed. Other examples of algorithms that employ reuse are from the predator-prey domain, where the predator prunes the search tree of a prior search to make it suitable for the current search, thereby saving the cost of re-expanding the pruned tree [26].

2.3. Anytime Path Planners

Anytime Path Planners are planners which can quickly develop a solution to the given problem and, if given more computation time, iteratively improve the plan quality. Anytime algorithms are desirable for many domains as they allow for metareasoning to make online tradeoffs between solution quality and planning time [27]. A naïve way to construct an anytime planner is to run a standard planner with parameters which trade solution optimality for a runtime improvement (e.g. A* heuristic inflation), and then iteratively re-run the planner with tighter bounds if computation time remains [28]. While this first plan generation is often fast, successive iterations grow increasingly slow due lack of information reuse. Anytime planners that instead reuse information from prior searches are generally faster at generating successive searches [29, 30, 31].

There exists other, non A*-like anytime path planners which also leverage reuse techniques, such as RRT* [10], which finds a feasible solution and then, given more time, repeatedly improves it by further sampling the space and updating the tree with cheaper intermediate nodes when applicable, converging to the optimal solution in the limit. Reuse and bounded search techniques can also be combined to further speed anytime search [32, 33].

2.4. Multiagent Path Planners

Multiagent Path Planners are planners designed to solve the MAPF problem. Prior work on these planners falls into two major classes: decoupled search and global search. Decoupled search operates by planning for each agent serially, reserving the location and time for each step of the plan, forcing following agent plans to avoid these reservations. This technique is common in both path planning and planning in general [21, 34, 35, 36, 37, 38]. Global search treats the MAPF problem as a single, large meta-agent search problem, and attempts to employ techniques that leverage the substructure of the problem to speed the search [39, 20, 40, 41, 42].

M* is a state-of-the-art A*-like global solver for optimal and ϵ-suboptimal MAPF problems [20]. It operates by first finding an optimal policy in the individual configuration space of each agent, and then combining these policies into a one dimensional search space embedded in joint configuration space. When agent-agent collisions are detected, the search space is locally expanded in joint space to allow for coupled planning for only the agents involved. In domains where agent-agent collisions are sparse, the dimensionality of these projections is low, thereby allowing M* to quickly solve the MAPF problem.

There is also work on bridging the gap between global and decoupled planning to exploit collision sparsity, such as Conflict Based Search (CBS) [43] and its ϵ-suboptimal counterpart [44]. CBS is a state-of-the-art non A*-like planner which builds a conflict graph, adds constraints to each agent, and replans in each agent's individual space, allowing for planning space to grow exponentially in the number of conflicts rather than the number of agents.

2.5. Anytime Multiagent Path Planners

Anytime Multiagent Path Planners combine techniques from Anytime Planning and Multiagent Planning to iteratively build higher quality solutions to the MAPF problem. Recent work by L. Cohen et. al. [19] introduced the first anytime MAPF solver, Anytime Focal Search (AFS). AFS works by maintaining a *focal* list of states from the openlist whose f-value is at most a suboptimality factor larger than the smallest f-value in the openlist. It uses a large suboptimality factor to quickly find a solution, and then tightens the suboptimality bound as time allows, reusing search efforts while generating higher quality solutions.

3. Problem Statement and Definitions

As stated in Section 1, this work will focus on solving the MAPF problem for discrete domains. Importantly, while the provided definitions focus on *Classical MAPF* domains [45] for ease of notation, this paper's novel contributions do not fundamentally require all actions to be of unit cost and thus can be extended to MAPF$_R$ domains [46]. We provide a formal definition of the state space (Section 3.1), paths through the state space (Section 3.2), the properties that all heuristics must guarantee (Section 3.3), and how these relate to A*-like f-values (Section 3.4).

3.1. States and Neighbors

A state s is a set containing one or more *configurations* of agents, where a configuration represents a single agent, e.g. a single agent's position in the world. For a given state s, there exists an agent set α which contains exactly the agents involved in s. For a given agent configuration, there is a discrete set of adjacent configurations, where the transition to each of these adjacent configurations has an associated numerical *cost*, e.g. time taken to reach the adjacent configuration from the given configuration. We define a *neighbor* function $N(s)$ to map s to its set of *neighboring* states, the Cartesian product of the adjacent configurations for each agent configuration in s, with neighboring states that induce invalid behavior, e.g. agent-agent collisions during transitions or in final configuration, removed. We assume that *neighbor* is a mutual relation, i.e. $s \in N(s') \iff s' \in N(s)$. Each neighbor $s' \in N(s)$ has an associated *cost*, which is the sum of the adjacent configuration costs, defined to be $c(s, s')$. A state can also have some agents removed, forming a new state of a smaller cardinality, via a filter function $\phi(s, \alpha)$ over the agent set α such that, given s with an associated agent set α such that $\forall \alpha' \subseteq \alpha : |\alpha'| = |\phi(s, \alpha')|$

3.2. Paths

A path π is a sequence of neighboring states from a given start state b to a given goal state e with its length denoted $|\pi|$. The ith element in the sequence π, denoted π_i, must be a neighbor of π_{i-1}, i.e. $\pi_i \in N(\pi_{i-1})$. Paths have an associated *cost*, which is the cost of traversing from beginning to the end of the path, denoted $\|\pi\|$ such that $\|\pi\| := \sum_i c(\pi_{i-1}, \pi_i)$ and a path π is *optimal* from b to e if and only if $\nexists \pi' : \pi'_1 = \pi_1 = b \wedge \pi'_{|\pi'|} = \pi_{|\pi|} = e \wedge \|\pi'\| < \|\pi\|$

Similar to Section 3.1, paths can also have agents removed from each state, forming a new path of equal length but with each state of a smaller cardinality, via a filter function $\Phi(\pi, \alpha) := \pi'$: $\forall i : \pi'_i := \phi(\pi_i, \alpha)$.

3.3. Heuristics

The purpose of a heuristic is to estimate the cost of a given state to a given goal. Heuristics, denoted as $h(s, e)$ for a given state s and a goal e, have two major properties: admissibility and consistency. For convenience, we denote the true cost from state s to goal e to be $c^*(s, e)$.

Admissible. A given heuristic is admissible if it never overestimates the minimum cost to the goal, i.e. $\forall s : h(s, e) \leq c^*(s, e)$

Consistent. A given heuristic is consistent if it never decreases more than the true cost decreases, i.e. the *triangle inequality*, $\forall s, s' : c^*(s, s') + h(s', e) \geq h(s, e)$, holds. Note that, while consistency implies admissibility, the reverse is not true.

3.4. f-values, g-values, and h-values

A given state s has three *values* associated with it: an f-value, a g-value, and an h-value. Unlike A*, these values are explicitly parameterized by the start and goal states, as the start and goal will move through the lifetime of the search. A g-value indicates the known cost to travel to s from a given start, b, and is denoted $g(s, b)$. When the search is started, this cost is not known, and thus infinite, for all states but b, i.e. $(\forall s : s \neq b \implies g(s, b) = \infty) \wedge (g(b, b) = 0)$

Over the course of the search, relevant states will have their g-value updated to reflect their minimal cost from b. An h-value is an estimate of the cost to travel from s to a given goal e that uses the syntax of the provided heuristic, i.e. $h(s, e)$. An f-value is defined to be the sum of the g-value and h-value, i.e. $f(s, b, e) := g(s, b) + h(s, e)$. f-values are as a real value to provide a total ordering over states.

Similar to Section 3.1 and Section 3.2, f-values, g-values, and h-values can also be filtered by an agent set, where the g-value and h-value of a state is the sum of the g-value and h-value of its individual agents. More formally, given a start b, a goal e, and a state s with an associated agent set α:

$$\begin{aligned} &\forall \alpha', \alpha'' : \alpha = \alpha' \cup \alpha'' \wedge \alpha' \cap \alpha'' = \varnothing \iff \\ &g(s, b) = g(\phi(s, \alpha'), b) + g(\phi(s, \alpha''), b) \wedge \\ &h(s, e) = h(\phi(s, \alpha'), e) + h(\phi(s, \alpha''), e) \end{aligned} \tag{1}$$

3.5. Domain Sparsity

Domain sparsity is a measure of the number of agents involved in any agent-agent collision relative to the total number of agents. For random domains, sparsity can be numerically estimated by the ratio $\frac{\text{grid area}}{\text{number of agents}}$ such that domains with *low* sparsity will have many agents involved in a single collision, such as a narrow, heavily trafficked passage, while domains with *high* sparsity have few agents involved in a single collision. This sparsity property is useful as, in sparse domains, local collision resolutions are often of a much lower dimensionality than the full joint search space. High domain sparsity also implies that the set of all globally optimal solutions is of a high cardinality, due to fewer constaints on the global solution, and thus a globally optimal path that is very similar to the individually planned joint space is more likely to exist.

4. A*

The A* algorithm [5] is a combinatorial search algorithm that produces and utilizes data structures which uphold specific properties. We present a novel formalization of these properties which form the basis for our other contributions, prove that these properties are upheld by A*, and show these properties are sufficient to prove fundamental results about A*.

4.1. The A Algorithm*

The pseudocode for A* is provided in Algorithm 1 using the definitions from Section 3. Note that, for convenience, we define $\text{top}(O, b, e)$ to be the state with the smallest f-value, i.e. $\text{top}(O, b, e) := s \in O : \arg\min f(s, b, e)$.

Algorithm 1 A*

```
1:  function A*($b, e$)
2:      $O \leftarrow \{b\}$
3:      $g(b, b) = 0$
4:      $C \leftarrow \varnothing$
5:      while $O \neq \varnothing$ do
6:          $s \leftarrow \text{top}(O, b, e)$
7:          if $s = e$ then return UnwindPath($C, e, b$)
8:          $O \leftarrow O \setminus \{s\}$
9:          if $s \in C$ then continue
10:         $C \leftarrow C \cup \{s\}$
11:         for all $s' \in N(s)$ do
12:             $O \leftarrow O \cup \{s'\}$
13:             $g(s', b) \leftarrow \min(g(s', b), g(s, b) + c(s, s'))$
14:     return NOPATH
```

4.2. A Search Trees and Valid Search Tree Properties*

A* constructs *A* Search Trees.* These search trees are comprised of two sets of states, an open set O and a closed set C, which store the information used in a search to find shortest paths. For an A* Search Tree to be considered *valid* given a start b and a goal e, the following *Valid Search Tree Properties* (VSTPs) must hold:

$$b \in O \cup C \tag{2}$$

$$\forall s \in O \cup C : s \neq b \implies \exists s' : \begin{cases} s' \in C \\ s' = \arg\min_{s'' \in N(s)} g(s'', b) \\ g(s', e) + c(s', s) = g(s, e) \end{cases} \tag{3}$$

$$\forall s \in C : N(s) \subseteq O \cup C \tag{4}$$

$$\forall s \in O \cup C : \exists \pi : \pi_1 = b \wedge \pi_{|\pi|} = s \wedge \forall s' \in \pi : s' \neq s \implies s' \in C \tag{5}$$

$$\forall s \in C, \forall s' \in O : f(s, b, e) \leq f(s', b, e) \tag{6}$$

As a consequence of these definitions, the following properties hold for all A* Search Trees where VSTPs hold:

Theorem 1. Given a consistent heuristic and an optimal path π from b to e, the f-values along the path are non-decreasing, i.e. $\forall i : f(\pi_i, b, e) \leq f(\pi_{i+1}, b, e)$.

Theorem 2. $\forall s \in C, \forall s', \exists \pi : (\pi_1 = b \wedge \pi_{|\pi|} = s') \wedge (\|\pi\| + h(s', e) < f(s, b, e)) \implies s' \in C$

Theorem 3. Given a consistent heuristic and a valid A* Search Tree with a start b, we know the optimal path from b to all states in C.

Theorem 4. Given a consistent heuristic, A* produces valid A* Search Trees.

Proofs are provided in Appendix A.

5. Windowed Anytime Multiagent Planning Framework

We introduce a framework for anytime multiagent path finding with quality bounds called Windowed Anytime Multiagent Planning Framework (WAMPF), presented in Algorithm 2. As WAMPF is an anytime framework, valid but potentially suboptimal solutions are provided at the end of each iteration of RecWAMPF (Algorithm 2) via Line 16, and a terminal, optimal solution is provided via Line 15. The quality bound associated with each solution is computed by taking cost of the current path $\|\pi\|$ and dividing it by a lower bound on the optimal path cost c which is estimated by the cost of the optimal individual space plans, computed on Line 2.

Algorithm 2 Windowed Anytime Multiagent Planning Framework

```
1: procedure WAMPF
2:     $\pi \leftarrow$ optimal, independently planned paths for all agents
3:     $W \leftarrow \varnothing$
4:     RecWAMPF($\pi, W, \|\pi\|$)
5: procedure RecWAMPF($\pi, W, c$)
6:     for all $w \in W$ do
7:         $w, \pi \leftarrow$ GrowAndReplanIn($w, \pi$)
8:         if $\exists w' \in W : w' \cap w \neq \varnothing$ then
9:             $W, \pi \leftarrow$ PlanInOverlapWindows($w, W, \pi$)
10:    while FirstCollisionWindow($\pi$) $\neq \varnothing$ do
11:        $w \leftarrow$ FirstCollisionWindow($\pi$)
12:        $W, \pi \leftarrow$ PlanInOverlapWindows($w, W, \pi$)
13:    for all $w \in W$ do
14:        if ShouldQuit($w$) then $W \leftarrow W \setminus \{w\}$
15:    if $W = \varnothing$ then return $\left(\pi, \min(\frac{\|\pi\|}{c}, \epsilon)\right)$
16:    report $\left(\pi, \frac{\|\pi\|}{c}\right)$
17:    RecWAMPF($\pi, W, c$)
18: function PlanInOverlapWindows($w, W, \pi$)
19:    for all $w' \in W : w' \cap w \neq \varnothing$ do
20:        $w \leftarrow w \cup w'$
21:        $W \leftarrow W \setminus \{w'\}$
22:    $\pi \leftarrow$ PlanIn($w, \pi$)
23:    $W \leftarrow W \cup \{w\}$
24:    return $(W, \pi)$
```

5.1. WAMPF Overview

WAMPF operates by planning for each agent individually to form a joint space plan which may contain agent-agent collisions which require repair. Each collision is repaired in its temporal order of occurrence, and is performed inside a construct known as a *search window*. A search window is constructed by limiting the search area to a small set of states surrounding the collision and limiting the search to only the agents involved. The repair start is where the relevant agents, i.e. the agents involved in the collision, enter the window and the repair goal is where the relevant agents leave the window. After this repair process has been applied to all collisions, the resulting plan is globally valid, but potentially suboptimal. As more time allows, each window's search space is then increased, moving the repair's start earlier in the joint plan and the repair's goal later in the joint plan, allowing for more of the search space to be considered, thereby producing a higher quality repair. As repairs only consider a subset of the total agents, they can induce collisions between repairs. If two repairs produce a collision, then their search windows are merged together and their combined agents are jointly repaired in the merged window.

To generate first solutions, WAMPF leverages domain sparsity. In high sparsity domains, each collision will not involve many agents, thereby allowing for search windows to decompose the problem into distinct, lower dimensional repairs which are easier to solve.

To generate optimal solutions, WAMPF leverages the implications of domain sparsity. As discussed in Section 3.5, higher sparsity implies that a global solution is more likely to exist that is similar to the individually planned joint solution. In domains with high sparsity, WAMPF leverages this fact by moving the starts and goals of the repairs along the existing joint solution; if such a globally optimal solution exists, WAMPF will find it without having to explore solutions that deviate significantly from the individually planned path.

5.2. WAMPF Pieces

WAMPF has five pieces that every WAMPF-based planner implements:

Definition of a search window. As discussed in Section 5.1, a *search window* is a bound placed around agent-agent collisions that limits the search space of the collision repair to a contiguous subset of the entire search space. For a given path π and associated agent set α, a window encapsulates local agent-agent collisions involving $\alpha' \subseteq \alpha$. All windows have an associated start b and goal e which are defined to be in the joint space of α', and where the following properties must hold:

$$\forall a \in \alpha', \exists s_b \in \Phi(\pi, \alpha') : \phi(b, \{a\}) = \phi(s_b, \{a\}) \tag{7}$$

$$\forall a \in \alpha', \exists s_e \in \Phi(\pi, \alpha') : \phi(e, \{a\}) = \phi(s_e, \{a\}) \tag{8}$$

where s_b is the first state in $\Phi(\pi, \alpha')$ such that $s_b \in w$, and where s_e is the last state in $\Phi(\pi, \alpha')$ such that $s_e \in w$. In addition, for any given window, w, there exists a *successor* window, denoted $s(w)$, where $w_2 = s(w_1)$, such that $w_1 \subset w_2$. Finally, we can define how to merge window two windows. Given two windows w and w' associated with α and α', respectively, we define the union operator $\cup$ which merges the two windows to form w'' with an associated α'' such that the space of w and w' are encapsulated in w'' and $\alpha'' = \alpha \cup \alpha'$.

FirstCollisionWindow. The subroutine FirstCollisionWindow(π) is invoked on Line 10 and Line 11 of Algorithm 2. This subroutine determines the first agent-agent collision along a given path π and returns a window encapsulating that collision. If there is no agent-agent collision along π, it returns $\varnothing$.

PlanIn. The subroutine PlanIn(w, π) is invoked on Line 22 in Algorithm 2. The given path, π, has an associated agent set, α. The given window, w, has an associated agent set, α', where $\alpha' \subseteq \alpha$. This subroutine generates a path in w in the joint space of α', from w's b to e and inserts it into the relevant subset of π.

GrowAndReplanIn. The subroutine GrowAndReplanIn(w_1, π) is invoked on Line 7 of Algorithm 2. The given path, π, has an associated agent set, α, and the given window, w_1, has an associated agent set, α', where $\alpha' \subseteq \alpha$. This subroutine grows w_1 by replacing it with its successor, w_2, and generates a path in w_2 in the joint space of α', from w_2's b to e and inserts it into the relevant subset of π. The subroutine then returns w_2, π. GrowAndReplanIn(w_2, π) is guaranteed to only be invoked when PlanIn(w_2, π) or GrowAndReplanIn(w_1, π) have previously been invoked.

ShouldQuit. The subroutine ShouldQuit(w) is invoked on Line 14 in Algorithm 2. This subroutine is a predicate that determines if the given window w should no longer be grown. This can be due to time restrictions, iteration restrictions, or other intelligent termination conditions. One such condition is when the globally optimal solution is found for the agents α' in w, which is achieved when w grows large enough that it no longer restricts the search tree, thereby allowing for an unimpeded search to form a valid A* search tree from the global start to the global goal of α'.

5.3. WAMPF Properties

If the requirements outlined in Section 5.2 are met, then WAMPF produces a valid first solution and an optimal full solution.

Theorem 5. If we assume PlanIn and GrowAndReplanIn produce optimal solutions, a valid solution exists, and ShouldQuit(w) discards w when an optimal solution is found, then WAMPF will produce an optimal solution.

Theorem 6. If we assume PlanIn and GrowAndReplanIn produce optimal solutions, a valid solution exists, then WAMPF will produce a valid solution after a single invocation of RecWAMPF.

Proofs are provided in Appendix B.

6. Naïve Windowing A*

Naïve Windowing A* (NWA*) is a naïve planner that implements the WAMPF. It is an A*-based solver which does not re-use existing repair search information when solving for a new repair in a larger window. As we will show, NWA* is inefficient to use in practice, but it serves as a baseline to differentiate the strengths inherent to the WAMPF from the strengths of efficient implementations of the WAMPF.

6.1. NWA Overview*

NWA* generates a first solution by placing a search window around each collision, finding an optimal repair inside that window, and then incorporating that repair result into the global solution. While additional time remains, NWA* grows each search window, generates a new repair result from scratch, and inserts the repair result into the global solution.

6.2. WAMPF Implementations

Definition of a search window. In NWA*, a window is initially structured as a cube, formed as the set of states with an L_∞ norm of less than a constant *radius* around a collision point. If this formulation violates the constraints provided in Section 5.2, then the radius is increased until the constraints are satisfied. To generate a successor, each corner of the window is moved away from the center a fixed amount. To merge two windows, the most extreme values for each corner are used, thereby forming a rectangle in individual space that encompasses the two rectangles being merged, and with an agent set which is the union of the agent sets of the two windows being merged.

FirstCollisionWindow. This subroutine operates by tracing the individual agents in π along their route until a collision is detected, at which point a collision center is selected and the window formed.

PlanIn. This subroutine is implemented as an A* search in the joint space of the agent set of w from the associated b to e.

GrowAndReplanIn. This subroutine is implemented by invoking $\textsc{PlanIn}(s(w), \pi)$.

ShouldQuit. This subroutine is implemented by returning true if, in the last search of w, the search tree associated with w was not restricted by the window constraints, or if the computation time budget has been exceeded.

7. Expanding A*

Expanding A* (X*) is an efficient planner that implements the WAMPF. It is an A*-based solver which re-uses existing repair search information when solving for a new repair in a larger window. As we will show, X* is significantly more efficient to use in practice than NWA*, and state-of-the-art time to first solution along with state-of-the-art time to optimal solution.

7.1. X Overview*

X* generates a first solution using the same process as NWA*: by placing a search window around each collision, finding an optimal repair inside that window, and then incorporating that repair result into the global solution. While additional time remains, X* then employs a three stage transformation, shown in Figure 1, to grow each search window while reusing prior search information, thereby allowing it to avoid recomputing a large portion of the search tree used in generating the new solution, and inserts the updated repair result into the global solution.

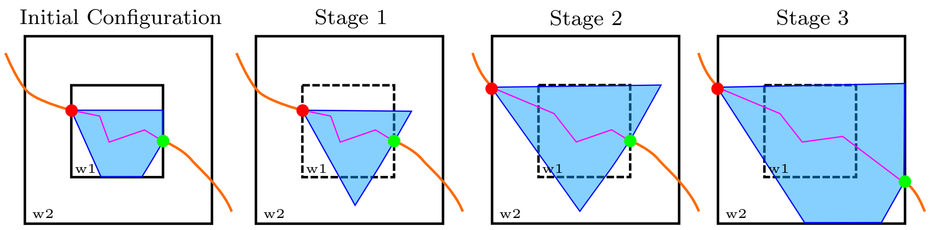


Figure 1: The three stages of X*'s grow and replan algorithm which allow it to save computation between searches. The red dot is the window start, the green dot is the window goal, the blue area is the search tree, the smaller box is the old window (w_1), the larger box is the new window (w_2), the purple path is the search solution, and the orange lines are non-colliding joint space paths. Initial Configuration to Stage 1 removes the restriction of the smaller window in the search from the old start to old goal, Stage 1 to Stage 2 moves the start from the old start to the new start, and Stage 2 to Stage 3 moves the goal from the old goal to the new goal.

7.2. X Search Trees and Valid Search Tree Properties*

For each window w, X* maintains an associated A*-style Search Tree, comprised of an open set, a closed set, f-values, g-values, and h-values. X* also introduces two new features: 1) a new data structure called an *out of window set*, X, a set which holds neighbor states which would otherwise be placed into O but were outside of w in the prior search, and 2) the notion of a *closed cost* associated with a state s and a start b, denoted $\Im(s, b)$, which was the $g(s, b)$ when s was last expanded. This extended A* Search Tree (Section 4.2) will be referred to as an *X* Search Tree.* X* Search Trees are carried forward between searches of an associated window and are initialized like A* Search Trees when a new window is created, but with X initialized to $\varnothing$ and closed cost values initialized, like g-values, to ∞.

Section 4.2 outlines VSTPs for A* Search Trees. We form a very similar set of properties for X* Search Trees to form X* VSTPs, or XVSTPs:

$$b \in O \cup C \tag{9}$$

$$\forall s \in O \cup C : s \neq b \implies \exists s' : \begin{cases} s' \in C \\ s' = \arg\min_{s'' \in N(s)} g(s'', b) \\ g(s', e) + c(s', s) = g(s, e) \end{cases} \tag{10}$$

$$\forall s \in C : N(s) \subseteq O \cup C \cup X \tag{11}$$

$$\forall s \in O \cup C : \exists \pi : \pi_1 = b \wedge \pi_{|\pi|} = s \wedge \forall s' \in \pi : s' \neq s \implies s' \in C \tag{12}$$

$$\forall s \in C, \forall s' \in O : f(s, b, e) \leq f(s', b, e) \tag{13}$$

$$\forall s \in X : s \notin w \wedge \exists s' : s \in N(s') \wedge s' \in C \tag{14}$$

$$\forall s \in C \cup O : s \in w \tag{15}$$

$$\forall s \in C : g(s, b) \geq \Im(s, b) \tag{16}$$

7.3. WAMPF Implementations

Three of X*'s five key implementations are identical to NWA* (Section 6.2); however, the other two make use of the guarantees provided by the WAMPF regarding the ordering of PlanIn and GrowAndReplanIn calls on successor windows to improve efficiency.

PlanIn. This subroutine is implemented as an A* search in the joint space of w from the associated b to e with two additions: 1) during state expansion, neighbors in w are placed in O, while neighbors not in w are placed in X (satisfying Property 11, a relaxation of Property 4, as well as Property 14 and Property 15), and 2) when a state is placed into C, its closed value is set to its g-value (satisfying Property 16). We know that A* produces valid A* Search Trees (Theorem 4), thereby satisfying Property 9 through Property 13, and these two additions ensure that the search trees generated are valid X* Search Trees.

GrowAndReplanIn. This subroutine uses the search tree of the previous search in the current search via the transformation shown in Figure 1. The algorithm is presented in Algorithm 3, and its supporting procedures are presented in Algorithms 4 – 7. GrowAndReplanIn begins with an O, C, and X that were populated from an earlier invocation of PlanIn or GrowAndReplanIn in w_1, as guaranteed in the definition of GrowAndReplanIn in Section 5.

7.3.1. Analysis of GrowAndReplanIn

A line-by-line breakdown of GrowAndReplanIn (Algorithm 3) is as follows:

- *Line 3:* This line grows the window by selecting its successor to be reasoned about by Stage1, Stage2, and Stage3.
- *Line 4:* Stage1 is designed to take these data structures and update them to contain the same information as if PlanIn were instead invoked with the b and e of w_1, but run with the constraints of w_2.

Algorithm 3

```
1: function GrowAndReplanIn($w_1, \pi$)
2:     do
3:         $w_2 \leftarrow s(w_1)$
4:         Stage1
5:         Stage2
6:         $\pi' \leftarrow$ Stage3
7:         $w_1 \leftarrow w_2$
8:     while $\pi' =$ NOPATH
9:     Replace section of $\Phi(\pi, \alpha)$ from $w_2$'s $b$ to $e$ with $\pi'$
10:    return $(w_2, \pi)$
```

- *Line 5:* Stage2 is designed to take these data structures and update them to contain the same information as if PlanIn were instead invoked with the b of w_2, the e of w_1, and the constraints of w_2.
- *Line 6:* Stage3 is designed to take these data structures and update them to contain the same information as if PlanIn were instead invoked in w_2 with the associated b and e, ultimately producing an optimal path from b to e in w_2, or `NOPATH` if no valid path exists in w_2.
- *Line 7:* This line performs bookkeeping to allow for retry in the event `NOPATH` is returned by the call on Line 6.
- *Line 8:* This loop is designed to handle the case where no solution is found. This is caused if there is no global solution, or if, due to the additional constraints of w, there is no solution to the local repair. This is handled by growing the window and rerunning the three stages.
- *Line 9:* This line inserts the repaired path π' into the global path π for the agents relevant to the search in w_1 and w_2, α.
- *Line 10:* This line returns the grown and searched window w_2 and the freshly repaired global joint path π.

7.3.2. Analysis of A Search Until*

AStarSearchUntil is a helper function designed to take X* Search Trees that violate specific properties and repair them to meet those properties. To do this, it runs a variant of A* until the minimal f-value of any state in O meets or exceeds the given threshold, f_{max}. The two primary differences between AStarSearchUntil and standard A* are 1) the addition of a mechanism for re-expanding a state if it was placed into C with a higher g-value than its current g-value (Line 5), and 2) the addition of neighbors not in w to X (Line 9).

Theorem 7. Given an X* Search Tree from b to e in w where $\forall s \in C : f(s, b, e) \leq f_{max}$ and all the XVSTPs hold except for:

Algorithm 4 A* Search Until

```
1: procedure A*SearchUntil($O, C, X, w, f_{max}$)
2:     while $f(\text{top}(O, b, e), b, e) < f_{max}$ do
3:         $s \leftarrow \text{top}(O, b, e)$
4:         $O \leftarrow O \setminus \{s\}$
5:         if $s \in C \wedge \Im(s, b) \leq g(s, b)$ then continue
6:         $C \leftarrow C \cup \{s\}$
7:         $\Im(s, b) \leftarrow g(s, b)$
8:         $O \leftarrow O \cup \{n \mid n \in N(s) : n \in w\}$
9:         $X \leftarrow X \cup \{n \mid n \in N(s) : n \notin w\}$
10:        for all $n \in N(s)$ do $g(n, b) \leftarrow \min(g(n, b), g(s, b) + c(s, n))$
```

- Property 13, which fails to hold for some $s \in O : f(s, b, e) < f_{max}$
- Property 16, which fails to hold for some $s \in C : \Im(s, b) + h(s, e) < f_{max} \wedge s \in O$

A*SearchUntil produces a valid X* Search Tree from b to e in w where

$$\forall s \in O : f(s, b, e) \geq f_{max} \tag{17}$$

$$\forall s \in C : f(s, b, e) \leq f_{max} \tag{18}$$

Proof 1. We know that, upon termination of A*SearchUntil, $\forall s \in O : f(s, b, e) \geq f_{max}$ holds as that is the terminating conditions of the loop (Line 2).

Lemma 1.1 (Upon termination of A*SearchUntil, Property 13 holds). We know that, upon termination, Property 13 will hold as A*SearchUntil will proceed until $\forall s \in O : f(s, b, e) \geq f_{max}$, removing all states with an f-value less than f_{max} from O (Line 4) and adding them to C (Line 6). In addition, we know from the givens that all violating states have an f-value less than f_{max} and thus will be processed by A*SearchUntil. □

Lemma 1.2 (Upon termination of A*SearchUntil, Property 16 holds). We know that, upon termination, Property 16 will hold as A*SearchUntil will proceed until $\forall s \in O : f(s, b, e) \geq f_{max}$. We know that states where Property 16 does not hold must be in O and have an f-value less than f_{max}, so they will be re-expanded (Line 5), allowing their closed cost to be properly set (Lines 6 – 10). In addition, we know from the givens that all violating states have an f-value less than f_{max} and thus will be processed by A*SearchUntil. □

From Lemma 1.1 we know Property 13 will hold upon termination and from Lemma 1.2 we know Property 16 will hold. The rest of the XVSTPs hold for the same reasons they hold for A* in Theorem 4 and its extensions in PlanIn in Section 7.3. □

Algorithm 5 Stage 1

```
1: procedure STAGE1
2:     $O \leftarrow O \cup \{s \mid s \in X : s \in w_2\}$
3:     $X \leftarrow \{s \mid s \in X : s \notin w_2\}$
4:     A*SEARCHUNTIL$(O, C, X, w_2, f(e_1, b_1, e_1))$
```

7.3.3. Analysis of Stage 1

STAGE1 is an algorithm designed to accept a valid X* Search Tree in window w_1 for start b_1 and e_1, and produce a valid X* Search Tree in window w_2 for start b_1 and goal e_1.

Theorem 8. Given a valid X* Search Tree from b_1 to e_1 in w_1 where

$$\forall s \in O : f(s, b_1, e_1) \geq f(e_1, b_1, e_1) \tag{19}$$
$$\forall s \in C : f(s, b_1, e_1) \leq f(e_1, b_1, e_1) \tag{20}$$

STAGE1 produces a valid X* Search Tree from b_1 to e_1 in w_2 where

$$\forall s \in O : f(s, b_1, e_1) \geq f(e_1, b_1, e_1) \tag{21}$$
$$\forall s \in C : f(s, b_1, e_1) \leq f(e_1, b_1, e_1) \tag{22}$$
$$\nexists s \in X : s \in w_2 \tag{23}$$

Proof 2. We do a line-by-line analysis of STAGE1:

- *Lines 2 – 3:* As we are converting the search from w_1 to w_2, removing all states from $s \in X : s \in w_2$ and adding them to O ensures that Property 14 holds for $w = w_2$ as well as ensures that Property 23 holds. However, doing so potentially violates Property 13, as it may be that $\exists s \in C, \exists s' \in X, \exists \pi' : \pi'_0 = b_1 \wedge \pi'_{|\pi'|} = e_1 \wedge s' \in \pi' \wedge \|\pi'\| < g(s, b_1)$.
- *Line 4:* We invoke A*SEARCHUNTIL on the valid X* Search Tree. Note that as per Property 22, $f(e_1, b_1, e_1)$ upperbounds the f-values for any states in C. Thus, while Property 13 does not hold, the X* Search Tree meets the preconditions for A*SEARCHUNTIL to generate a valid X* Search Tree (Theorem 7), and thus STAGE1 terminates with a valid X* Search Tree.

7.3.4. Analysis of Stage 2

STAGE2 is an algorithm designed to accept a valid X* Search Tree in the window w_2 for the start b_1 to the goal e_1 and produce a valid X* Search Tree in the window w_2 for start b_2 and goal e_1.

Theorem 9. Given a valid X* Search Tree from b_1 to e_1 in w_2 where

$$\forall s \in O : f(s, b_1, e_1) \geq f(e_1, b_1, e_1) \tag{24}$$
$$\forall s \in C : f(s, b_1, e_1) \leq f(e_1, b_1, e_1) \tag{25}$$

Algorithm 6 Stage 2

```
1: procedure Stage2
2:     $\pi' \leftarrow \Phi(\pi, \alpha)$ between $b_2$ and $b_1$
3:     for all $s \in O \cup C$ do
4:         $g(s, b_2) \leftarrow g(s, b_1) + \|\pi'\|$
5:     for all $s \in C$ do $\Im(s, b_2) \leftarrow \Im(s, b_1) + \|\pi'\|$
6:     for all $s \in \pi'$ do
7:         $C \leftarrow C \cup \{s\}$
8:         $\Im(s, b_2) \leftarrow g(s, b_2)$
9:         $O \leftarrow O \cup \{n \mid n \in N(s) : n \in w_2\}$
10:        $X \leftarrow X \cup \{n \mid n \in N(s) : n \notin w_2\}$
11:        for all $n \in N(s)$ do $g(n, b_2) \leftarrow \min(g(n, b_2), g(s, b_2) + c(s, n))$
12:    A*SearchUntil$(O, C, X, w_2, f(e_1, b_1, e_1) + \|\pi'\|)$
```

Stage2 produces a valid X* Search Tree from b_2 to e_1 in w_2 where

$$\forall s \in O : f(s, b_2, e_1) \geq f(e_1, b_2, e_1) \tag{26}$$

$$\forall s \in C : f(s, b_2, e_1) \leq f(e_1, b_2, e_1) \tag{27}$$

Proof 3. We know that we are given a valid X* Search Tree for b_1 and e_1 in w_2, but this tree is not nessicarily a valid X* Search Tree for b_2 and e_1 in w_2. In paticular, we know that if $b_2 \neq b_1$, there is no guarantee that Property 9 and thus Property 12 hold. We do a line-by-line analysis of Stage2:

- *Line 2:* This line does not modify any of the X* Search Tree, and thus the XVSTPs continue to fail to hold. Note that π' is the path between b_2 and b_1. We know that π' is collision free and optimal as it was not part of w_1 nor is it part of any other window; if it were, w_2 would overlap with that window and thus have been merged with that window.

- *Line 4:* We know that $\forall s \in O \cup C : g(s, b_1)$ is the minimal cost to go from b_1 to s. We know that $\|\pi'\|$ is the minimal cost to go from b_2 to b_1, and so we know that the minimal cost to go from b_2 to s is upperbounded by $g(s, b_1) + \|\pi'\|$. Thus, after this line, we know that $\forall s \in O \cup C : g(s, b_2)$ is an exact estimate or an overestimate of the cost from b_2 to s. In addition, as this is an addition of a scalar constant to all g-values, this does not disrupt the relative ordering of g-values and thus the XVSTPs with the exception of Property 9 and Property 12 must continue to hold.

- *Line 5:* We know that, using the same reasoning as Line 4, we know $\forall s \in C : \Im(s, b_2)$ is an exact estimate or an overestimate of the cost from b_2 to s.

- *Lines 6 – 11:* These lines are expanding all states along π'; note that they are identical to A*SearchUntil's state expansion (Algorithm 4 Lines 7

– 10). As we know π'' is a minimal cost path from b_2 to b_1, we know that the path along π to each of the intermediary states must also be minimal cost. These expansions could violate Property 13 and Property 16, but they ensure that Property 9 and Property 12 hold.

- *Line 12:* We know from above that all of the XVSTPs but Property 13 and Property 16 hold. Similarly, we know that $f(e_1, b_1, e_1) + \|\pi'\|$ upperbounds the f-values for any states in C. Thus the X* Search Tree meets the preconditions for A*SearchUntil to generate a valid X* Search Tree (Theorem 7), and thus Stage1 terminates with a valid X* Search Tree.

$\square$

7.3.5. Analysis of Stage 3

Algorithm 7 Stage 3

```
procedure Stage3
    if $e_2 \in C$ then return UnwindPath($C, e_2, b_2$)
    while $O \neq \varnothing$ do
        $s \leftarrow \text{top}(O, b_2, e_2)$
        if $s = e_2$ then return UnwindPath($C, e_2, b_2$)
        $O \leftarrow O \setminus \{s\}$
        if $s \in C$ then continue
        $C \leftarrow C \cup \{s\}$
        $\Im(s, b_2) \leftarrow g(s, b_2)$
        $O \leftarrow O \cup \{n \mid n \in N(s) : n \in w_2\}$
        $X \leftarrow X \cup \{n \mid n \in N(s) : n \notin w_2\}$
        for all $n \in N(s)$ do $g(n, b_2) \leftarrow \min(g(n, b_2), g(s, b_2) + c(s, n))$
    return NOPATH
```

Stage3 is an algorithm designed to accept a valid X* Search Tree in the window w_2 for the start b_2 to the goal e_1 and produce a valid X* Search Tree in the window w_2 for start b_2 and goal e_2.

Theorem 10. Given a valid X* Search Tree from b_2 to e_1 in w_2 where

$$\forall s \in O : f(s, b_2, e_1) \geq f(e_1, b_2, e_1) \quad (28)$$
$$\forall s \in C : f(s, b_2, e_1) \leq f(e_1, b_2, e_1) \quad (29)$$

Stage3 produces a valid X* Search Tree from b_2 to e_2 in w_2 where

$$\forall s \in O : f(s, b_2, e_2) \geq f(e_2, b_2, e_2) \quad (30)$$
$$\forall s \in C : f(s, b_2, e_2) \leq f(e_2, b_2, e_2) \quad (31)$$

Proof 4. As we know from Section 5.2, e_2 is associated with w_2, which is the successor of w_1 with its associated e_1. Thus, we know that, as per the definition of a successor window, the minimal cost to reach e_2 from b_2 must be greater or equal to the minimal cost to reach e_1 from b_2. Thus, we have a valid X* Search Tree that either has already expanded e_2 because $f(e_2, b_2, e_1) = f(e_1, b_2, e_1)$, in which case a minimal cost path to e_2 can immediately be unwound from C (Line 2), or $f(e_2, b_2, e_1) > f(e_1, b_2, e_1)$, in which case running a modified version of A* will allow for a minimal cost path to be found.

Section 7.3 outlines modified A* planner that produces valid X* Search Trees; Lines 3 – 13 is that modified A* planner sans its initialization. We know as a consequence of the XVSTPs that if a goal is changed, it will the heuristic estimate for each state, thereby influencing the f-value minimal states in O and thus the direction of future search. The only of XVSTPs that relies on f-value is Property 13, but the same consistent heuristic is used in the search to e_2 as was used in e_1, and thus the triangle inequality that consistent heuristics must follow requires Property 13 to hold. Thus, Stage3 produces a valid X* Search Tree from b_2 to e_2 in w_2 which upholds Property 30 and Property 31. □

8. Empirical Results

First, we compare X*'s performance against its baselines (Section 8.1) and against state-of-the-art anytime and optimal MAPF solvers on random problems (Section 8.2). This leads us to explore the relationship between sparisty of the domain and X*'s runtime (Section 8.3) as well as establish the relationship between this sparsity and the components of X* that dominate its runtime (Section 8.4), leading to an investigation of how X* parameter selection impacts this relationship (Section 8.5). Finally we compare X* against state-of-the-art MAPF solvers on standard benchmarks (Section 8.6).

All planners were implemented in C++. X*, NWA*, and A* were implemented by the authors of this paper[1], AFS was implemented by its associated authors, CBS was implemented by a third party[2], and M* was implemented by its associated authors[3], with the Operator Decomposition version used for all benchmarks. All runtime measurements were performed on a dedicated computer with an Intel i7 CPU and access to 60GB of DDR4 RAM. Any trial that exceeded the memory limit was recorded as a timeout.

8.1. X Versus Baselines*

X* operates by restricting the initial repair search space, quickly finding an optimal solution in this restricted search space, then relaxing the restriction and repeating the process until a globally optimal solution is found. While this approach provides WAMPF's anytime property, it also has incurs computational overhead, even from planners like X* which perform reuse between repair searches.

[1]Source code available at `https://github.com/kylevedder/libMultiRobotPlanning`
[2]Source code available at `https://github.com/whoenig/libMultiRobotPlanning/`
[3]Source code available at `https://github.com/gswagner/mstar_public/`

Planner	**First Solution**	**Optimal Solution**	**RecWAMPF iteration at median A* runtime**
X*	0.0158, 0.01658, 0.0199	0.4539, 0.4595, 0.5225	6 / 9
NWA*	0.0158, 0.01648, 0.0241	1.4011, 1.4353, 1.7327	4 / 9
A*	–	0.2535, 0.2623, 0.2903	–

Table 1: X*, NWA*, and A* run on a 20×20 grid with an agent starting on the center of each edge and with a goal on the center of the opposite edge to demonstrate the overhead of WAMPF-style window growth compared to A*. Each result is reported in seconds as a 95% confidence interval in *2.5th percentile, 50th percentile, 97.5th percentile* format.

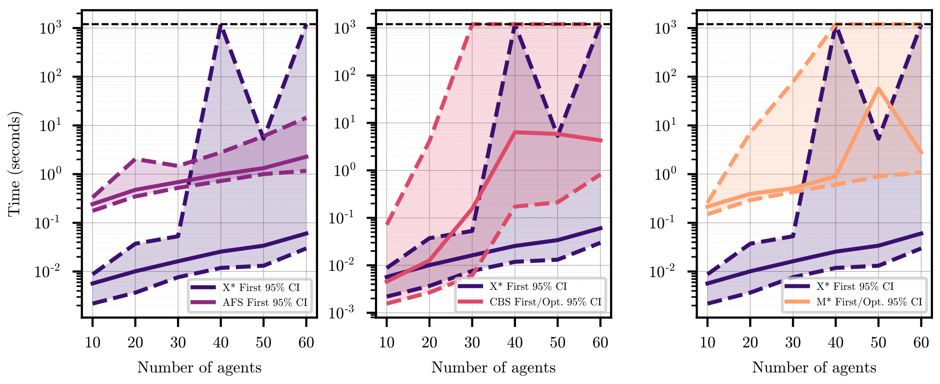
To demonstrate this overhead, we run X*, NWA* and A* on a 20×20 four-connected grid scenario with an agent starting on the center of each edge and with a goal on the center of the opposite edge, thereby inducing a single collision in the center of the scenario. While A* will directly solve for an optimal solution, X* and NWA* will produce multiple solutions.

The results are presented in Table 1, with 95% confidence intervals over 30 trials. Due to identical structure of their initial solution generation, NWA* and X* have nearly identical performance for time to first solution and outperform A*'s time to optimal solution by over an order of magnitude. Due to the window overhead, X* takes approximatly 1.5 times longer than A* to produce an optimal solution, having finished 5 of the needed 9 window expansions when A* terminates, and NWA* takes more than an order of magnitude longer longer than A* to produce an optimal solution due to a lack of search re-use, having finished 3 of the needed 9 window expansions when A* terminates.

8.2. X Performance On Random Grids*

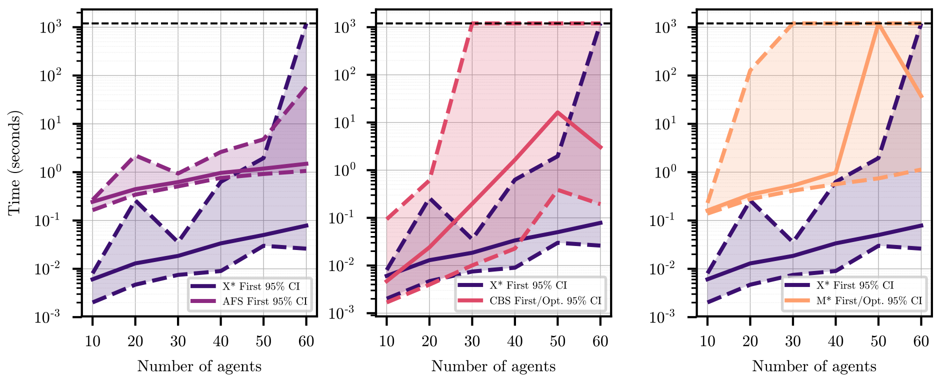
In order to put X*'s performance in context with the state-of-the-art, we compare its performance against AFS, the only other anytime MAPF solver, and against CBS and M*, two state-of-the-art optimal MAPF solvers. All experiments are performed on a 100×100 four-connected grid with random starts and goals. Agent counts and the percentage of total cells blocked are varied. Both time to first solution on 1% (Figure 2), 5% (Figure 3), and 10% (Figure 4) blocked grid cells and time to optimal solution on 1% (Figure 5), 5% (Figure 6), and 10% (Figure 7) blocked grid cells are presented. Increasing the percentage of blocked cells provides insight into how each planner performs under decreasing sparsity and increasing the number of agents provides insight into how each planner performs under increasing agents and increasing sparsity.

In all experiments, X* significantly outperforms AFS in first solution generation as X* directly exploits domain sparsity to perform low dimensional local repairs while AFS performs a joint suboptimal search, allowing X* to scale with the number of collisions rather than the number of agents. For optimal solution generation, X* outperforms AFS on low numbers of agents and underperforms AFS on higher numbers of agents, and this is difference is more pronounced in the more heavily occupied domains. This difference is once again due to domain sparsity; for lower numbers of agents or more open domains, the domain is more



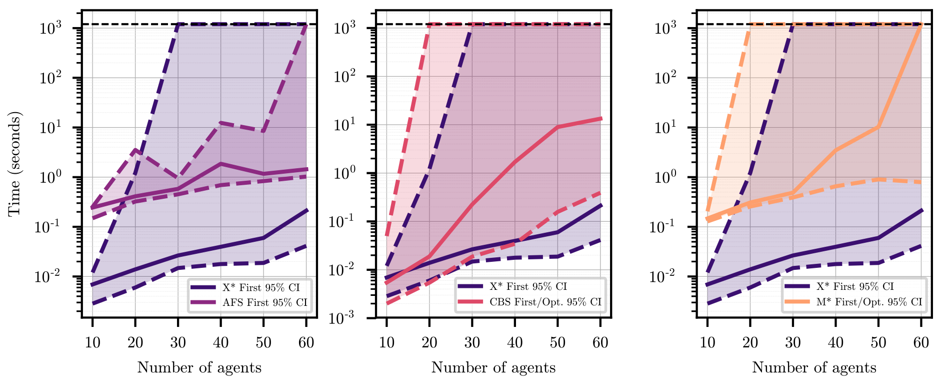
(a) X* vs AFS (b) X* vs CBS (c) X* vs M*

Figure 2: 95% confidence intervals of time to first solution for X*, AFS, CBS, and M*. For each agents count, 30 trials are run, each with a 20 minute timeout, with each trial run on a randomly generated 1% blocked 100×100 four-connected grid.



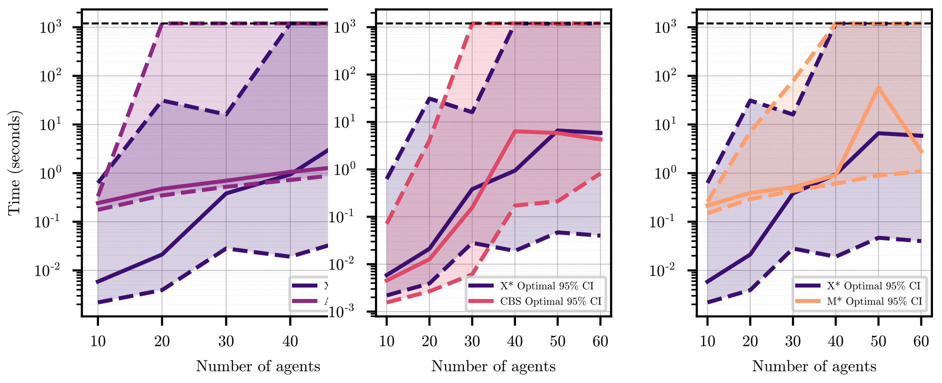
(a) X* vs AFS (b) X* vs CBS (c) X* vs M*

Figure 3: 95% confidence intervals of time to first solution for X*, AFS, CBS, and M*. For each agents count, 30 trials are run, each with a 20 minute timeout, with each trial run on a randomly generated 5% blocked 100×100 four-connected grid.



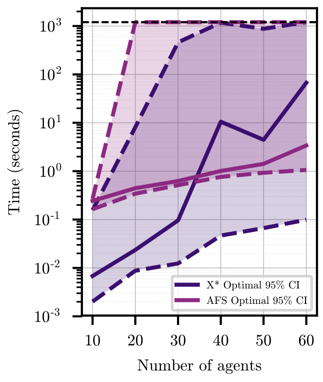
(a) X* vs AFS (b) X* vs CBS (c) X* vs M*

Figure 4: 95% confidence intervals of time to first solution for X*, AFS, CBS, and M*. For each agents count, 30 trials are run, each with a 20 minute timeout, with each trial run on a randomly generated 10% blocked 100 × 100 four-connected grid.

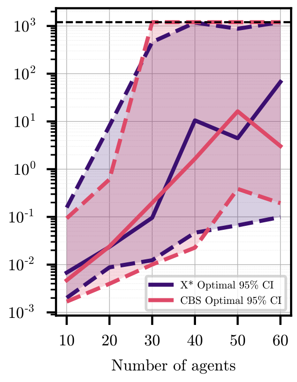


(a) X* vs AFS (b) X* vs CBS (c) X* vs M*

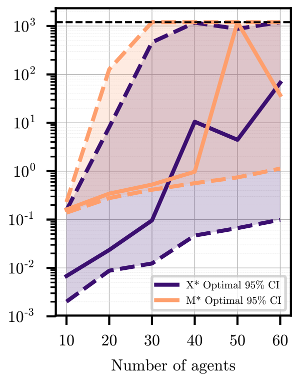
Figure 5: 95% confidence intervals of time to optimal solution for X*, AFS, CBS, and M*. For each agents count, 30 trials are run, each with a 20 minute timeout, with each trial run on a randomly generated 1% blocked 100 × 100 four-connected grid.



(a) X* vs AFS

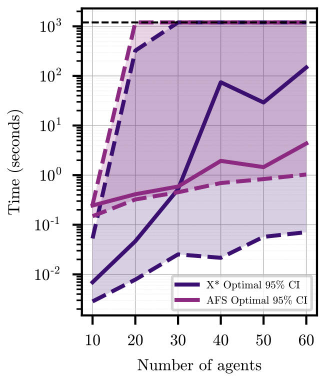


(b) X* vs CBS

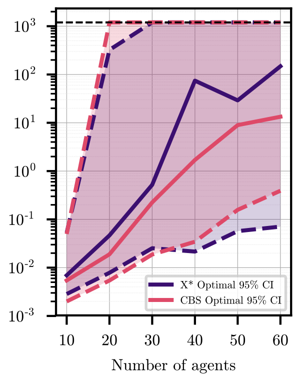


(c) X* vs M*

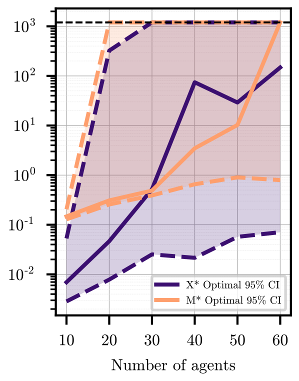
Figure 6: 95% confidence intervals of time to optimal solution for X*, AFS, CBS, and M*. For each agents count, 30 trials are run, each with a 20 minute timeout, with each trial run on a randomly generated 5% blocked 100 × 100 four-connected grid.



(a) X* vs AFS



(b) X* vs CBS



(c) X* vs M*

Figure 7: 95% confidence intervals of time to optimal solution for X*, AFS, CBS, and M*. For each agents count, 30 trials are run, each with a 20 minute timeout, with each trial run on a randomly generated 10% blocked 100 × 100 four-connected grid.

sparse, so there are few collisions and thus X* has few search windows that potentially need to be merged, allowing X* to scale with the number of collisions. As sparsity decreases, more of its windows need to be merged together and the global solution becomes increasingly dissimilar to the individually planned joint path, reducing the effectiveness of X*'s search re-use technique.

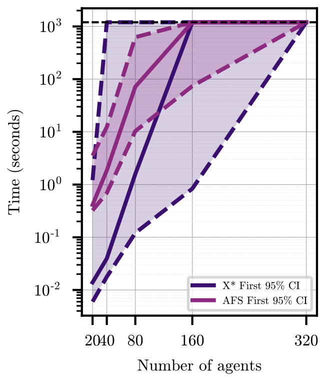
In all experiments, X* performs better than CBS in first solution generation, particularly as the number of agents grows; this is unsurprising, as CBS produces optimal paths while X*'s first solution need only be valid, but the performance gap is very small for low numbers of agents. This is due to the fact that both X* and CBS directly leverage domain sparsity and thus scale with the number of collisions. For optimal solution generation, X* underperforms CBS in the less sparse domains despite both directly exploiting domain sparsity. X*'s worse performance can be attributed to the overhead of its re-use technique as discussed in Section 8.1 as well as the increased likelihood of window merges, reducing the effectiveness of X*'s search re-use technique.

In all experiments, X* significantly outperforms M* in first solution generation. It is unsurprising that X* outperforms M* for first solution generation, as M* produces optimal paths while X*'s first solution need only be valid. For optimal solution generation, X* outperforms M* on low numbers of agents and underperforms M* on higher numbers of agents. This difference appears invariant to percentage of blocked grid cells, and this is due to the high degree of similarity in how X* and M* exploit domain sparsity. M* operates on interaction sets, which are similar to X* windows, but are intelligently sized and incorporate the proper number of agents to fully resolve a collision. X* performs worse on higher numbers of agents due to the overhead of its re-use technique as discussed in Section 8.1 as well as the increased likelihood of window merges, reducing the effectiveness of X*'s search re-use technique.

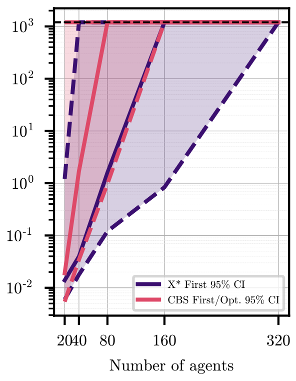
8.3. X Runtime Versus Sparsity of Domain*

X* is designed to exploit *sparsity* of agent-agent collisions in order to quickly develop a suboptimal but valid solution as well as produce its globally optimal solution. As a consequence, it is expected that X* will scale well when the number of agents in a domain increase, but the level of sparsity stays the same. To validate this expectation, we run X* on varying sized four-connected grids with a 10% obstacle density and constant $\frac{\text{grid area}}{\text{agent count}}$ ratio of 500. We also run CBS, AFS, and M* on the same domains to provide a frame of reference. Time to first solution is presented in Figure 8 and time to optimal solution is presented in Figure 9.

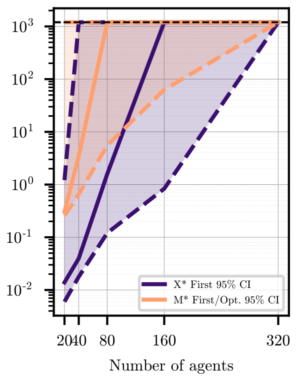
For time to first solution, X*'s median time is faster than any other planner. CBS performs the best of the other planners on 20 agents, but its interval lowerbound closely matches X*'s median for the rest of the agents. Interestingly, for all agent counts that do not result in a timeout, AFS's median time is approximately two orders of magnitude slower than X*'s median, despite both being anytime. Finally, X*'s interval lowerbound is significantly below the lowerbound of any other planner, particularly for the 160 agent case.



(a) X* vs AFS

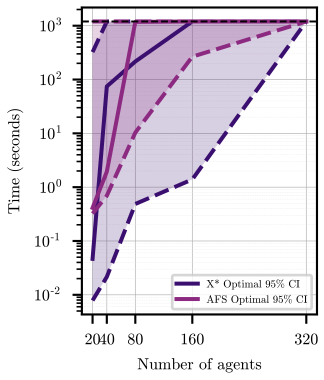


(b) X* vs CBS

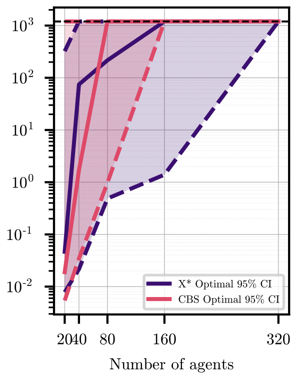


(c) X* vs M*

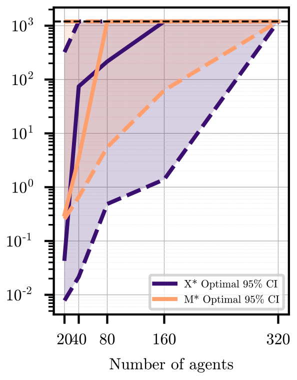
Figure 8: 95% confidence intervals of time to first solution for X* vs AFS, CBS, and M*. For each agents count, 30 trials are run, each with a 20 minute timeout, with each trial run on with a constant $\frac{\text{grid area}}{\text{agent count}}$ ratio of 500 with a 10% obstacle density.



(a) X* vs AFS



(b) X* vs CBS



(c) X* vs M*

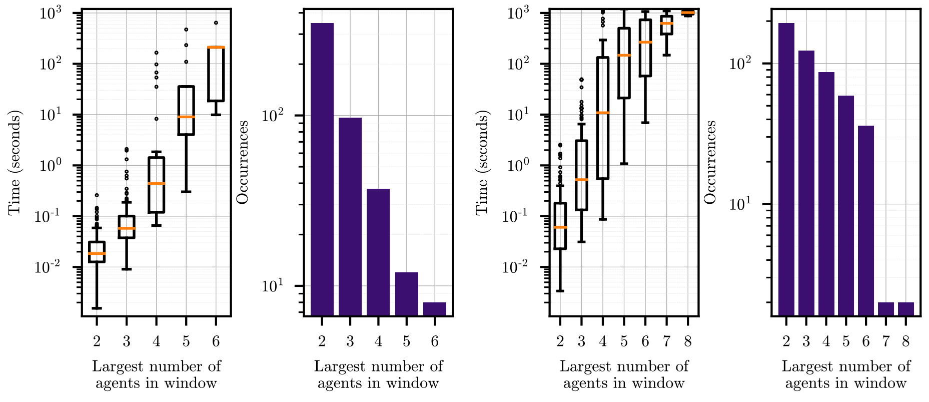
Figure 9: 95% confidence intervals of time to optimal solution for X* vs AFS, CBS, and M*. For each agents count, 30 trials are run, each with a 20 minute timeout, with each trial run on with a constant $\frac{\text{grid area}}{\text{agent count}}$ ratio of 500 with a 10% obstacle density.

For time to optimal solution, X* has a higher median runtime than the other planners for lower numbers of agents; however, for 80 agents, X*'s median runtime is below the timeout threshold while all other planners medians are at the timeout threshold. Furthermore, X*'s interval lowerbound is significantly below the lowerbound of every other planner, particularly for the 160 agent case.

Together, these findings suggest that, compared to state-of-the-art algorithms, X*'s approach scales well across domains with constant sparsity.

8.4. X Components Which Dominate Runtime*

In order to optimize X*, be it from an implementation perspective or a theoretical one, is important to understand which components dominate its runtime. X*'s runtime is dominated by PlanIn and GrowAndReplanIn, where the window searches with the highest number of agents dominate both time to first solution (Figure 10a) and time to optimal solution (Figure 10b). Fortunately, for random domains of various agents, as the number of agents involved in a window grows linearly, the number of occurrences of such a window decreases exponentially for both first solutions (Figure 10a) and optimal solutions (Figure 10b).

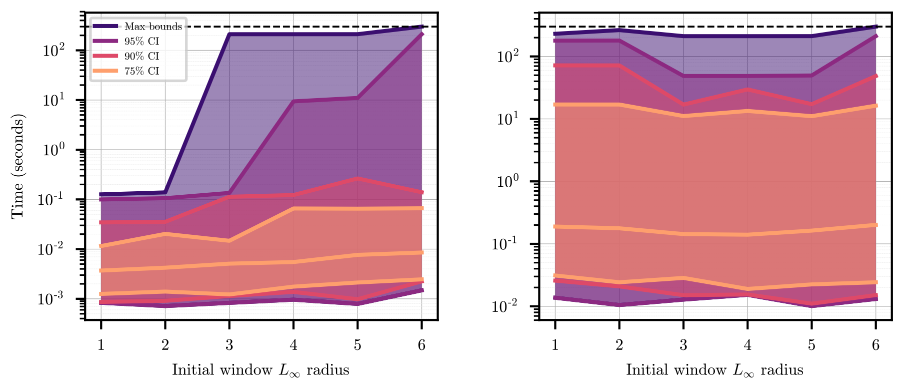


(a) First Solution (b) Optimal Solution

Figure 10: 95% confidence intervals of time to first and optimal solutions for X* vs maximum number of agents involved in any single repair. Run across 30 trials of 20 to 60 agents on 100×100 four-connected grids with 1%, 5%, and 10% obstacle density.

8.5. X Window Selection Impact on Runtime*

As shown in Section 8.4, window dimensionality dominates runtime. As such, selecting the proper initial window size to repair a search in order to minimize window merges is an important factor in X*'s first solution generation performance. Figure 11a shows the impact of the initial window radius parameter on X*'s time to first solution; unsurprisingly, smaller window radii translate to a decreased likelihood of requiring window merges and thus faster first solution generation.



(a) First Solution (b) Optimal Solution

Figure 11: Confidence intervals of time to first solution and time to optimal solution vs initial window radius for X*. Run across 30 trials of 30 agents on 100 × 100 four-connected grids with 5% obstacle density.

However, smaller window radii can *increase* time to first solution in some cases. Shown in Figure 11b, an initial window radius of 1 or 2 result in 90% and 95% interval bounds that are almost an order of magnitude higher than the bounds produced by initial window radii of 3, 4, and 5. The root cause of this significant performance degradation is the expansion of states during a small window search which would not be expanded by a fresh search in a larger window. As discussed in Section 5.1, WAMPF's window growing technique repeatedly moves the window search's goal along the existing joint solution; however, this can lead to the expansion of states in a smaller search window which would not be expanded by a fresh search in a larger window, as moving the goal changes the value and relative ordering of state's h-value. As such, these unnecessary expansions earlier in X*'s search will add states to O to be expanded which would never be considered by a search that initially had a larger window. The exact radius values for which performance degrades changes across scenarios as a consequence of the structure of the domain, making this analysis important for practitioners who care about time to optimal solution.

8.6. X On Common Benchmarks*

Finally, to characterize X* in more structured domains, we compare it against AFS, CBS, and M* on several standard benchmark domains[4] presented in Table 2, namely `den520d`, `brc202d`, `lak303d`, `ht_mansion_n`, `ost003d`, and `w_woundedcoast`. All 25 random instances are tried with 50 agents, and each instance is run with a 300 second timeout.

Due to the sparsity of the domains, CBS and X* demonstrated very fast solution generation while M* often took two orders of magnitude more time to

[4]Benchmarks available at `https://movingai.com/benchmarks/mapf/index.html`

Scenario	X*	AFS	CBS	M*
den520d	0.0021, 0.0026, 0.0033	10.6765, 12.0885, 13.2662	–	–
	0.0021, 0.0027, 0.0092	10.6765, 12.0885, 13.2662	0.0017, 0.0024, 0.0057	4.0300, 4.2496, 4.6512
brc202d	0.0031, 0.0038, 0.0044	7.3286, 8.1243, 11.7510	–	–
	0.0032, 0.0050, 0.2200	7.3287, 8.1310, 300	0.0029, 0.0037, 300	4.4278, 4.6910, 5.1294
lak303d	0.0018, 0.0023, 0.0044	2.5149, 2.6335, 2.9173	–	–
	0.0018, 0.0052, 1.1907	2.5149, 2.6335, 300	0.0015, 0.0023, 0.0209	1.5467, 1.5907, 1.8268
ht_mansion_n	0.0016, 0.0021, 0.0042	0.7010, 0.7354, 0.7660	–	–
	0.0017, 0.0035, 0.0201	0.7010, 0.7357, 0.7660	0.0012, 0.0017, 0.0114	0.6987, 0.7301, 0.7976
ost003d	0.0018, 0.0022, 0.0225	2.0346, 2.1470, 2.2395	–	–
	0.0018, 0.0037, 1.0898	2.0346, 2.1470, 300	0.0013, 0.0018, 300	1.3746, 1.4160, 1.4953
w_woundedcoast	0.0061, 0.0104, 300	1.8082, 1.9448, 2.1081	–	–
	0.0064, 0.0180, 300	1.8083, 1.9466, 2.1081	0.0064, 0.0173, 0.0996	3.0273, 3.1426, 3.4299

Table 2: X*, CBS, AFS, and M* run on various standard benchmarks for 50 agents on all 25 provided random instances with a timeout of 300 seconds. Time to first solution reported in the first row and time to optimal solution reported in the second row. Each result is reported in seconds as a 95% confidence interval in *2.5th percentile, 50th percentile, 97.5th percentile* format.

solve the same problems and AFS often took twice as long as M*.

9. Future Work

X* uses standard A* to preform optimal window searches; if a suboptimal search technique such as AFS were used to admit suboptimal solutions inside a window, and this search tree could be grown using X*-style reuse, this approach may allow for even faster first solution generation. This investigation would also lend itself well to better exploring ϵ-suboptimal WAMPF.

In addition, there is room for further exploration of window size and shape; in this work, we worked with square and rectangular windows because they are easy to reason about and performed better than rasterized spheres, but there may exist more complex window shapes that are better suited to WAMPF.

Finally, we believe that further investigation into quantifying sparsity of MAPF domains would provide great insight into the fundamental nature of MAPF problem and allow for the development of ensemble based planners that can switch techniques based on the structure of the given problem or class of problems.

10. Acknowledgements

This work is supported in part by AFRL and DARPA under agreement #FA8750-16-2-0042, and NSF grant IIS-1724101. We would also like to thank Professor Sven Koenig and Liron Cohen for their AFS implementation, Wolfgang Hoenig for his implementation of CBS, and Glen Wagner for his implementation of M*.

11. Bibliography

References

[1] C. S. Ma, R. H. Miller, MILP optimal path planning for real-time applications, in: 2006 American Control Conference, 2006, pp. 6 pp.–.

[2] J. Berger, A. Boukhtouta, A. Benmoussa, O. Kettani, A new mixed-integer linear programming model for rescue path planning in uncertain adversarial environment, Computers & Operations Research 39 (2012) 3420–3430.

[3] W. N. N. Hung, X. Song, J. Tan, X. Li, J. Zhang, R. Wang, P. Gao, Motion planning with Satisfiability Modulo Theories, in: 2014 IEEE International Conference on Robotics and Automation (ICRA), 2014, pp. 113–118.

[4] V. Nguyen, P. Obermeier, T. C. Son, T. Schaub, W. Yeoh, Generalized Target Assignment and Path Finding Using Answer Set Programming, in: SOCS, 2017.

[5] M. Hard, N. Nilsson, B. Raphael, A Formal Basis for the Heuristic Determination of Minimum Cost Paths, in: IEEE Transactions on Systems Science and Cybernetics SSC4., 1968.

[6] J. Reif, Complexity of the Generalized Movers Problem, in: J. Schwartz, J. Hopcroft, M. Sharir (Eds.), Planning, Geometry, and Complexity of Robot Motion, Ablex Publishing Corp, 1987, Ch. 11, pp. 267–281.

[7] D. Harabor, A. Grastien, Online Graph Pruning for Pathfinding on Grid Maps, in: Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence, AAAI'11, AAAI Press, 2011, pp. 1114–1119.

[8] S. Karaman, E. Frazzoli, Sampling-based Algorithms for Optimal Motion Planning, The International Journal of Robotics Research 30 (7) (2011) 846–894.

[9] L. E. Kavraki, P. Svestka, J. C. Latombe, M. H. Overmars, Probabilistic roadmaps for path planning in high-dimensional configuration spaces, in: IEEE Transactions on Robotics and Automation, Vol. 12, IEEE, 1996, pp. 566–580.

[10] S. Karaman, E. Frazzoli, Sampling-based algorithms for optimal motion planning, in: International Journal of Robotics Research, Vol. 30, 2011, pp. 846–894.

[11] J. Hopcroft, J. Schwartz, M. Sharir, On the Complexity of Motion Planning for Multiple Independent Objects; PSPACE - Hardness of the "Warehouseman's Problem", in: The International Journal of Robotics Research, 1984, pp. 76–88.

[12] D. Harbor, S. Koenig, N. Sturtevant, AAMAS 2019 Tutorial on Heuristic Search (2019).

[13] A. Felner, M. Barer, N. R. Sturtevant, J. Schaeffer, Abstraction-Based Heuristics with True Distance Computations, in: SARA, 2009.

[14] J. Yu, S. M. LaValle, Planning optimal paths for multiple robots on graphs, in: 2013 IEEE International Conference on Robotics and Automation, 2013, pp. 3612–3617.

[15] P. Surynek, Towards optimal cooperative path planning in hard setups through satisfiability solving, in: PRICAI, 2012.

[16] P. Surynek, A. Felner, R. Stern, E. Boyarski, Efficient sat approach to multi-agent path finding under the sum of costs objective, 2016.

[17] R. Barták, J. Svancara, On SAT-Based Approaches for Multi-Agent Path Finding with the Sum-of-Costs Objective, Symposium on Combinatorial Search (SoCS).

[18] E. Erdem, D. G. Kisa, U. Öztok, P. Schüller, A General Formal Framework for Pathfinding Problems with Multiple Agents, in: AAAI, 2013.

[19] L. Cohen, M. Greco, H. Ma, C. Hernández, A. Felner, T. K. S. Kumar, S. Koenig, Anytime Focal Search with Applications, in: IJCAI, 2018.

[20] G. Wagner, Subdimensional Expansion: A Framework for Computationally Tractable Multirobot Path Planning, Ph.D. thesis, The Robotics Institute Carnegie Mellon University (2015).

[21] D. Silver, Cooperative Pathfinding, in: Proceedings of the First AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, AIIDE'05, AAAI Press, 2010, pp. 117–122.

[22] K. Vedder, J. Biswas, X*: Anytime Multiagent Path Planning With Bounded Search, in: E. Elkind, M. Veloso (Eds.), Autonomous Agents and Multiagent Systems, 2019.

[23] A. Kushleyev, M. Likhachev, Time-bounded lattice for efficient planning in dynamic environments, 2009 IEEE International Conference on Robotics and Automation (2009) 1662–1668.

[24] S. Aine, M. Likhachev, Truncated Incremental Search, Artificial Intelligence 234 (C) (2016) 49–77.

[25] S. Koenig, M. Likhachev, D* Lite, in: Proceedings of the AAAI Conference of Artificial Intelligence, AAAI, 2002, pp. 476–483.

[26] X. Sun, W. Yeoh, S. Koenig, Generalized Fringe-Retrieving A*: faster moving target search on state lattices, in: AAMAS, 2010.

[27] J. Svegliato, S. Zilberstein, Adaptive Metareasoning for Bounded Rational Agents, in: CAI-ECAI Workshop on Architectures and Evaluation for Generality, Autonomy and Progress in AI (AEGAP), Stockholm, Sweden, 2018.

[28] R. Zhou, E. A. Hansen, Multiple sequence alignment using A*, in: Proceedings of the AAAI Conference of Artificial Intelligence, 2002.

[29] M. Likhachev, G. Gordon, S. Thurn, ARA*: Anytime A* with Provable Bounds on Sub-Optimality, in: Advances in Neural Information Processing Systems 16: Proceedings of the 2003 Conference, 2003.

[30] S. Aine, P. P. Chakrabarti, R. Kumar, AWA*-a Window Constrained Anytime Heuristic Search Algorithm, in: Proceedings of the 20th International Joint Conference on Artifical Intelligence, IJCAI'07, 2007, pp. 2250–2255.

[31] R. Natarajan, M. S. Saleem, S. Aine, M. Likhachev, H. Choset, A-MHA*: Anytime Multi-Heuristic A*, in: Twelfth Annual Symposium on Combinatorial Search, SoCS'19, 2019.

[32] S. Aine, M. Likhachev, Anytime Truncated D* : Anytime Replanning with Truncation, in: Proceedings of the Sixth Annual Symposium on Combinatorial Search, SOCS 2013, Leavenworth, Washington, USA, July 11-13, 2013., 2013.

[33] V. Narayanan, M. Phillips, M. Likhachev, Anytime Safe Interval Path Planning for dynamic environments, 2012 IEEE/RSJ International Conference on Intelligent Robots and Systems (2012) 4708–4715.

[34] M. Erdmann, T. Lozano-Pérez, On multiple moving objects, Algorithmica 2 (1) (1987) 477.

[35] K. Kant, S. W. Zucker, Toward Efficient Trajectory Planning: The Path-Velocity Decomposition, The International Journal of Robotics Research 5 (3) (1986) 72–89.

[36] S. Leroy, J. P. Laumond, T. Simeon, Multiple Path Coordination for Mobile Robots: A Geometric Algorithm, in: Proceedings of the 16th International Joint Conference on Artificial Intelligence - Volume 2, IJCAI'99, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999, pp. 1118–1123.

[37] M. Saha, P. Isto, Multi-Robot Motion Planning by Incremental Coordination, 2006 IEEE/RSJ International Conference on Intelligent Robots and Systems (2006) 5960–5963.

[38] M. Crosby, A. Jonsson, M. Rovatsos, A Single-agent Approach to Multiagent Planning, in: Proceedings of the Twenty-first European Conference on Artificial Intelligence, ECAI'14, IOS Press, Amsterdam, The Netherlands, The Netherlands, 2014, pp. 237–242.

[39] M. R. K. Ryan, Exploiting Subgraph Structure in Multi-Robot Path Planning, J. Artif. Intell. Res. 31 (2008) 497–542.

[40] T. Scott Standley, Finding Optimal Solutions to Cooperative Pathfinding Problems., Vol. 1, 2010.

[41] A. Felner, M. Goldenberg, G. Sharon, R. Stern, T. Beja, N. R. Sturtevant, J. Schaeffer, R. C. Holte, Partial-Expansion A* with Selective Node Generation, in: AAAI, 2012.

[42] M. Goldenberg, A. Felner, R. Stern, G. Sharon, N. Sturtevant, R. C. Holte, J. Schaeffer, Enhanced Partial Expansion A*, J. Artif. Int. Res. 50 (1) (2014) 141–187.

[43] G. Sharon, R. Stern, A. Felner, N. R. Sturtevant, Conflict-based search for optimal multi-agent pathfinding, Artificial Intelligence 219 (2015) 40 – 66.

[44] M. Barer, G. Sharon, R. Stern, A. Felner, Suboptimal Variants of the Conflict-based Search Algorithm for the Multi-agent Pathfinding Problem, in: Proceedings of the Sixth International Symposium on Combinatorial Search, 2014.

[45] R. Stern, N. R. Sturtevant, D. Atzmon, T. Walker, J. Li, L. Cohen, H. Ma, T. K. S. Kumar, A. Felner, S. Koenig, Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks, Symposium on Combinatorial Search (SoCS) (2019) 151–158.

[46] J. Lang (Ed.), Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden, ijcai.org, 2018.

Appendix A. A* Search Tree Theorem Proofs

Theorem 1. Given a consistent heuristic and an optimal path π from b to e, the f-values along the path are non-decreasing, i.e. $\forall i : f(\pi_i, b, e) \leq f(\pi_{i+1}, b, e)$.

Proof 5. Consider an optimal path π from start b to goal e; π_{i+1} is the successor to π_i, and so we know from Property 3:

$$\begin{aligned} f(\pi_{i+1}, b, e) &= g(\pi_{i+1}, b) + h(\pi_{i+1}, e) \\ &= g(\pi_i, b) + c(\pi_i, \pi_{i+1}) + h(\pi_{i+1}, e) \\ &\geq g(\pi_i, b) + h(\pi_i, e) \\ &= f(\pi_i, b, e) \end{aligned}$$

Theorem 2. $\forall s \in C, \forall s', \exists \pi : (\pi_1 = b \wedge \pi_{|\pi|} = s') \wedge (\|\pi\| + h(s', e) < f(s, b, e)) \implies s' \in C$

Proof 6. Assume $\forall s \in C, \exists s' \notin C, \exists \pi : (\pi_1 = b \wedge \pi_{|\pi|} = s') \wedge (\|\pi\| + h(s', e) < f(s, b, e))$. We know that if π exists, then an optimal path to s' exists, i.e. $\exists \pi^*, \forall \pi' : b = \pi^*_1 = \pi'_1 \wedge e = \pi^*_{|\pi^*|} = \pi'_{|\pi'|} \wedge \|\pi^*\| \leq \|\pi'\|$. We know from Property 6 that $s' \notin O$, and so π^* must start in $C \cup O$ (Property 2), have one or more states in O in order to have a state not in $O \cup C$ (Property 4). We know that f-values of states along π^* are non-decreasing (Theorem 1), and so we know that $\exists s'' \in \pi^* : s'' \in O \wedge f(s'', b, e) < f(s, b, e)$ (Property 3). However, as $s \in C$ and $s'' \in O$, this violates Property 6. Thus we have reached a contradiction and the proof is complete. □

Theorem 3. Given a consistent heuristic and a valid A* Search Tree with a start b, we know the optimal path from b to all states in C.

Proof 7. $\forall s \in C$, we know that there exists a path π from b to s (Property 5) and thus we can unwind an *optimal* path from s to b by repeatedly selecting the neighbor with the lowest cost to b. We know that g-values estimate the lowest cost to start (Lemma 7.1) and thus we can unwind an optimal path from s to b by repeatedly selecting the neighbor with the lowest g-value, and then reversing that path to produce an optimal path from b to s. $\square$

Lemma 7.1 (g-values are minimal cost to b $\forall s \in C$). Assume $\exists s^{\neg *} \in C$ for which g-value is not minimal. This implies that there is an optimal path π to get from b to $s^{\neg *}$ such that $\|\pi\| < g(s^{\neg *}, b)$. We know that $\forall s \in \pi : s \in C$, as we are given that $s^{\neg *} \in C$, we know from Theorem 1 that f-values of states along a path increase monotonically, and we know from Theorem 2 that $\forall s \in \pi : s \in C$. However, one of the states in π must violate Property 3, as $\|\pi\| < g(s^{\neg *}, b)$ requires that $\exists i : g(\pi_{i+1}, b) - g(\pi_i, b) > c(\pi_i, \pi_{i+1})$, and from above $\forall s \in \pi : s \in C$. Thus, this assumption reaches a contradiction, and thus the proof is complete. $\square$

Theorem 4. Given a consistent heuristic, A* produces valid A* Search Trees.

Proof 8. A* operates by repeatedly *expanding* states, a term which refers to the process of removing a state from O (Line 8), and, if applicable (Line 9), placing it in C (Line 10) and placing its neighbors in O (Lines 11 – 13).

Base case. For Lines 2 – 4, we know the VSTPs hold. Property 2 holds from Line 2. Property 3 holds from Lines 2 – 4. Property 4 is vacuously true as $C = \varnothing$ (Line 4). Property 5 is trivially true as the path from b to b is trivial. Property 6 is vacuously true as $C = \varnothing$ (Line 4).

Inductive case. We assume for the inductive hypothesis that the statement holds for all previous iterations of the loop body on Lines 6 – 13, and we set about proving it for this iteration.

- *Line 6:* no change to O or C occurs, so the VSTPs hold.
- *Line 7:* if $s = e$ holds, A* exits and the VSTPs hold; otherwise, no change to O or C occurs, so the VSTPs hold.
- *Line 8:* s is removed from O. If $s = b$, this violates Property 2, and if $s \neq b$ this violates Property 4.
- *Line 9:* if $s \in C$ holds, then if $s \neq b$ then Property 4 we now know holds, and if $s = b$, then we now know that Property 2 holds, and thus the loop completes and the VSTPs hold; otherwise, no change to O or C occurs, so either Property 2 or Property 4 do not hold.
- *Line 10:* as discussed regarding Line 8 and the fact that $s \notin C$, adding s to C ensures Property 2 holds in the case $s = b$. Property 6 holds for

s as it was selected via top(O, b, e) (Line 6) which by definition selects a state with a minimal f-value from O, so $\nexists s' \in O : f(s', b, e) < f(s, b, e)$. Property 5 holds for s when added to C as it held when s was in O. However, Property 4 may not hold for s as its neighbors may not been added to O or C.

- *Lines 11 – 13:* These lines are designed to ensure that Property 4 holds for s by adding all of its neighbors, $N(s)$, to O, and ensures that Property 3 holds for all of those neighbor states. In particular, $s' \in O$ is guaranteed after Line 12. Either there was no previously known way to get to s', in which case the minimal cost way is via s, or there was a prior known way; the min (Line 13) ensures that the lower of the two costs, either via s, or via a different state, is recorded as the g-value of s', such that Property 3 holds. We know Property 5 holds $\forall n \in N(s)$ as they can be reached via the path to s.

  Property 6 holds for:

  - $\forall n \in N(s) : n \in C$ as n already has an optimal g-value (Theorem 3)
  - $\forall n \in N(s) : n \in O$ as f-value of n is the min of its previous f-value (which must respect Property 6) and $g(s, b) + c(s, n) + h(n, e) \geq f(s, b, e)$ and s has the minimal f-value for all states in O
  - $\forall n \in N(s) : n \notin O \cup C$ as f-value of n is $g(s, b) + c(s, n) + h(n, e) \geq f(s, b, e)$ and s has the minimal f-value for all states in O

□

Appendix B. WAMPF Proofs

Theorem 5. If we assume PlanIn and GrowAndReplanIn produce optimal solutions, a valid solution exists, and ShouldQuit(w) discards w when an optimal solution is found, then WAMPF will produce an optimal solution.

Proof 9.

Lemma 9.1 (Optimal merged paths are optimal). Given two paths, π for agent set α and π' for agent set α', where π and π' are optimal, $\alpha \cap \alpha' = \varnothing$, and π and π' do not collide with each other, then if π and π' are joined to produce π'', from Section 3.2 it follows that $\|\pi''\| = \|\pi\| + \|\pi'\|$, and thus $\|\pi''\|$ is optimal.

Proof by contradiction: Consider a case where π'' constructed via the method above, is not optimal. That would imply that there exists another, optimal path with the same b and e, π''', such that:

$$\begin{aligned}
\|\pi'''\| &= \|\Phi(\pi''', \alpha)\| + \|\Phi(\pi''', \alpha')\| \\
&< \|\pi''\| \\
&= \|\Phi(\pi'', \alpha)\| + \|\Phi(\pi'', \alpha')\| \\
&= \|\pi\| + \|\pi'\| \\
&\Longrightarrow \\
\|\Phi(\pi''', \alpha)\| < \|\pi\| &\vee \|\Phi(\pi''', \alpha')\| < \|\pi'\|
\end{aligned}$$

which implies that π or π' are suboptimal, which violates the assumption that π and π' are optimal. □

Lemma 9.2 (Unrestricted window searches produce optimal paths). Given a joint path π and window w with an associated agent set α is used to repair $\Phi(\pi, \alpha)$, if w contains b and e associated with $\Phi(\pi, \alpha)$ and w did not constrain the search between b and e, then an optimal repair of $\Phi(\pi, \alpha)$ has been found.

We know from the definition of a window that if b and e associated with $\Phi(\pi, \alpha)$ are in w, then they are the b and e used by w. Thus, we know that as the given repair strategy produces an optimal solution in w between w's b and e, w's b and e are b and e of $\Phi(\pi, \alpha)$, and the search was not restricted by w, then this solution would be optimal even for an arbitrarily large w, and thus is a globally optimal path for $\Phi(\pi, \alpha)$. □

Lemma 9.3 (RecWAMPF always grows all windows). Given a set of windows W, all $w \in W$ will be enlarged by RecWAMPF to encompass more states.

At the start of each iteration of RecWAMPF, GrowAndReplanIn will be invoked $\forall w \in W$ (Lines $6 - 7$), by definition causing all windows to be grown, thereby upholding the claim. Some of these windows may be merged with existing windows (Line 9), resulting in a larger, merged windows (Line 20), thereby upholding the claim. Some of these windows may be merged with newly created windows, resulting in larger, merged windows (Line 12), thereby upholding the claim. □

When RecWAMPF is called, we know a given path is either:

1. Valid and globally optimal, with $W = \varnothing$
2. Invalid and at or below cost of globally optimal solution, with $W = \varnothing$
3. Valid and potentially globally suboptimal, with windows surrounding locally optimal repairs, i.e. $W \neq \varnothing$

We do an analysis of RecWAMPF in these three cases:

1. We invoke RecWAMPF with a valid and globally optimal solution and $W = \varnothing$. Lines $6 - 9$ are skipped, as $W = \varnothing$. Lines $10 - 12$ are skipped, as no collisions exist. Lines $13 - 14$ are skipped, as $W = \varnothing$. Finally, $W = \varnothing$, so $(\pi, 1)$ is returned with π unmodified (Line 15) and thus RecWAMPF returns π, having proved it's a globally optimal solution.

2. We invoke RecWAMPF with an invalid solution at or below joint optimal cost and $W = \varnothing$. Lines 6 – 9 are skipped, as $W = \varnothing$. Lines 10 – 12 create windows and locally repair each collision as they occur along π, merging windows if they overlap. When Lines 10 – 12 are complete, π is a valid but potentially globally suboptimal solution. If Lines 13 – 14 can prove that all windows produces local repairs that are globally optimal, then RecWAMPF returns π, having proven it's a globally optimal solution. Otherwise, RecWAMPF has produced a valid and potentially globally suboptimal solution with windows surrounding locally optimal repairs, the scenario handled by Case 3.
3. We invoke RecWAMPF with a valid and potentially globally suboptimal path π with windows surrounding locally optimal repairs. We know from Lemma 9.3 that these windows will continue to grow with each recursive invocation of RecWAMPF, any overlapping windows will be merged together (Lines 8 – 9), and any new collsions induced by repairs will be encapsulated by a new window and merged with any overlapping existing windows (Lines 10 – 12). Thus, we know in a finite number of recursive invocations of RecWAMPF, every window w, associated with an agent set α, will eventually contain b and e associated with $\Phi(\pi, \alpha)$ such that the window based repair between b and e is not constrained by w. Thus, we know from Lemma 9.2 that the globally optimal path for α from b to e has been proven to be found, and thus w can be removed from W by ShouldQuit (Lines 13 – 14). Thus, after a finite number of iterations, RecWAMPF will terminate and from Lemma 9.1 we know that the globally optimal solution has been found.

We know that RecWAMPF will only be invoked in the three cases:

1. π is composed of individually planned, optimal paths (Line 2), and it is collision free. $W = \varnothing$ (Line 3), and so it qualifies for Case 1. Case 1 always terminates after a single invocation of RecWAMPF, and π has been proved to be optimal.
2. π is composed of individually planned, optimal paths (Line 2), and it is *not* collision free. $W = \varnothing$ (Line 3), and so it qualifies for Case 2. Case 2 either terminates after a single invocation of RecWAMPF, and π has been proved to be optimal, or it invokes RecWAMPF in Case 3.
3. π is in the process of being repaired, making it potentially globally suboptimal, and it has an associated window set $W \neq \varnothing$. Case 3 either terminates and π has been proved to be optimal, or it again invokes Case 3.

□

Theorem 6. If we assume PlanIn and GrowAndReplanIn produce optimal solutions, a valid solution exists, then WAMPF will produce a valid solution after a single invocation of RecWAMPF.

Proof 10. This is a special case of Case 1 or Case 2 in Proof 9; as shown, either π generated on Line 2 is optimal, in which case WAMPF terminates with π as its solution, or π will be repaired to generate a valid solution. $\square$