X*: Anytime Multiagent Path Planning With Bounded Search

Kyle Vedder and Joydeep Biswas

University of Massachusetts Amherst, Amherst MA 01003, USA
{kvedder, joydeepb} @umass.edu

Abstract. Multi-agent planning in dynamic domains is a challenging problem: the size of the configuration space increases exponentially in the number of agents, and plans need to be re-evaluated periodically to account for moving obstacles. However, we have two key insights that hold in several domains: 1) conflicts between multi-agent plans often have *geometrically local* resolutions within a small repair window, even if such local resolutions are not globally optimal; and 2) the partial search tree for such local resolutions can then be iteratively improved over successively larger windows to eventually compute a globally optimal plan. Building upon these two insights, we introduce 1) a high level overview of an anytime multiagent planning framework called WAMPF, 2) a naïve implementation of WAMPF called NWA*, 3) an efficient implementation of WAMPF called X* which saves computation by reusing prior search information when improving a solution, and 4) Lazy Neighbor Evaluation, a novel approach to managing high cardinality neighbors.

1 Introduction

Constructing collision-free paths from a start to a goal in realtime is a problem faced by almost all robotic systems, from wheeled robots to grasping arms to flying drones. A challenging problem itself, there has been a wide variety of work devoted to quickly constructing such paths [10][11] and repairing them when environments change or obstacles move [12]. However, the problem of finding collision-free paths for *multiple* agents that also avoid colliding with one another, known as the Multiagent Planning Problem (MPP), presents another layer of difficulty. 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. As a result, this problem becomes intractable even when planning for only a handful of agents. In general, while the problem of finding a path for a single agent is NP-hard, solving the MPP for an arbitrary number of agents is PSPACE-hard [9].

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

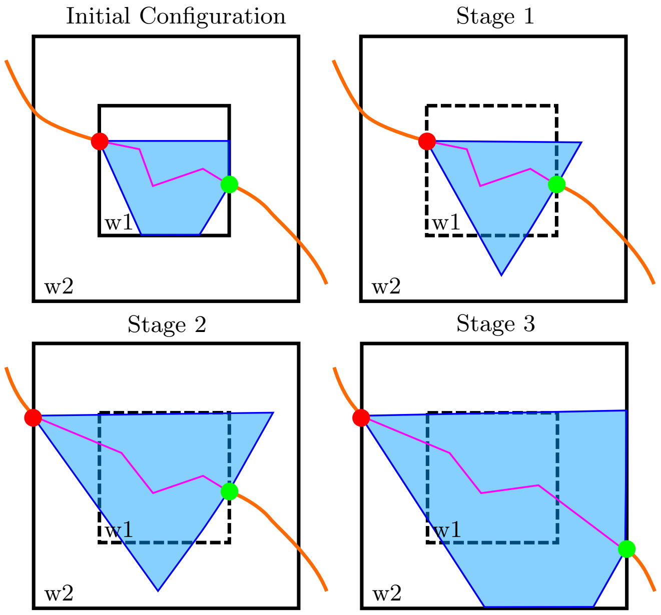

Fig. 1: Stages of `GrowAndReplanIn`. How X* reuses search information between windows.

In this work we focus on domains with an arbitrary number of agents but with "sparse" agent-agent interactions, where sparsity is a function of the number of agent-agent interactions relative to the total number of agents. This sparsity means that the number of agents in each interaction is lower than the total number of agents, and this fact can be exploited to speed solving.

We present four contributions in this paper. First, we present a planning framework which exploits the sparsity of these interactions in order to quickly solve each interaction locally, allowing for fast first solution generation. Second, we present a naïve implementation of this framework. Third, we present an efficient implementation of this framework that reuses information between searches to speed successive solution generation. Fourth, we present Lazy Neighbor Expansion which reduces memory overhead and improves runtime performance when neighbor cardinality is high. While the second, third, and fourth contributions focus on A*-like planners, the first contribution is applicable to a variety of search techniques. With these four contributions, we demonstrate a significant performance advantage over state-of-the-art optimal solvers for time to first solution while still providing competitive time to optimal solution.

2 Related Work

To put our contribution in the context of the state-of-the-art, we review existing approaches related to 1) bounded search, 2) search reuse, 3) anytime planning, 4) multiagent planning, and 5) anytime multiagent planning.

Bounded Search is a technique where artificial limits are placed on the search space. While bounds usually produce a suboptimal solution, they prevent planning for contingencies that are far away or unimportant, speeding solution generation. This bound can be enforced via the time domain such as with a time-bounded lattice [13], via depth of search such as Hierarchical Cooperative A* [18], or via restricted cost propagation such as Truncated D* Lite [3].

Search Reuse is a technique where information from a previous search is used to speed up the next search. One of the most famous of such algorithms, D* Lite [12], 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 [19].

Anytime 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 [20]. 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 [23]. 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 faster at generating successive searches [14,1].

There exists other, non A*-like anytime path planners which also leverage this concept of reuse, 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 [2,15].

Multiagent Planners are planners designed to solve the MPP. Prior work on these planners fall 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 [18] and planning in general [6]. Global search treats the MPP 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 [16,21].

M* is a state-of-the-art A*-like global solver for optimal and ϵ-suboptimal MPPs [21]. 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 low, thereby allowing M* to quickly solve the MPP.

There is also work on bridging the gap between global and decoupled planning to exploit this sparsity of interaction, such as Conflict Based Search (CBS) [17] and its ϵ-suboptimal counterpart [4]. 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.

Anytime Multiagent Path Planners combine techniques from Anytime Planning and Multiagent Planning to iteratively build higher quality solutions to the MPP. Very recent work by L. Cohen et. al. [5] introduced the first anytime MPP 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 and generating higher quality solutions.

3 Contributions

In this work we present 1) a framework for performing anytime multiagent path planning (WAMPF) 2) Naïve Window A* (NWA*), a naïve implementation of WAMPF, 3) Expanding A* (X*), an efficient implementation of WAMPF, and 4) Lazy Neighbor Expansion, a method for speeding up A* and A*-like planners when the cardinality of neighbors is high.

3.1 Windowed Anytime Multiagent Planner Framework

Algorithm 1 presents the general anytime sparse interaction algorithm for multi-agent path planning called the Windowed Anytime Multiagent Planner Framework (WAMPF). WAMPF first computes individual space plans for each agent. It then invokes the recursive planner RecWAMPF. RecWAMPF operates by growing and replanning all existing windows (Lines 6 - 9), being sure to merge together any newly grown windows which interfere with other windows (Lines 8 - 9), and then finding and repairing any collisions that initially existed or were introduced by the window growth (Lines 10 - 12), thereby guaranteeing a collision free path at the end of every invocation. The four subroutines provided by the implementation of every planner using the RecWAMPF framework are 1) FirstCollisionWindow, 2) GrowAndReplanIn, 3) PlanIn, and 4) ShouldQuit.

FirstCollisionWindow(Π) returns a window, w, which is an artificial constraint placed on the search space of the interacting set of agents which

Algorithm 1 Windowed Anytime Multiagent Planner

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

encapsulates an agent-agent interaction in Π. This constraint allows for a focused search to quickly find a joint path free of that interaction. In addition, for an arbitrary window w_1, it has a successor, w_2, where $w_1 \subset w_2$.

ShouldQuit(w) 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 optimal solution is found inside w, which is achieved when w grows large enough that it no longer restricts the search tree of interacting agents, thereby allowing for an unimpeded search to form a valid A* search tree from the start to the goal.

PlanIn(w, Π) plans inside a given window w, replacing the section of Π that travels through w with a path planned from scratch.

GrowAndReplanIn(w, Π) grows and then plans inside a given window w, updating of the section of Π that travels through w. It has the option of reusing the information from the prior search in the smaller un-grown window during its planning process.

3.2 Naïve Window A*

Naïve Window A* (NWA*) is a naïve implementation of WAMPF. It provides the four needed subroutines of WAMPF.

FirstCollisionWindow(Π) is implemented by modeling a window as a cube in joint space defined by an interaction center and a "radius" measured by the L_∞ norm from the center. It possesses a start b and goal e, each in the joint space of interacting agents. These states sit either on a face of the window or inside it, with the position of each individual agent defined to be that agent's individually defined path as it enters and exits the window in individual space, or its individual start or goal.

ShouldQuit(w) is implemented by quitting if in the last iteration, the search tree associated with w was not restricted by the window constraints.

PlanIn(w, Π) is implemented as a joint space A* search with the window constraints.

GrowAndReplanIn(w, Π) is implemented as a growth via an increase in the window radius, updating of b and e, and then a joint space A* search within the window constraints, with no efficient search reuse.

3.3 Expanding A*

Expanding A* (X*) is an efficient implementation of WAMPF. It is implemented identically to NWA* but with a GrowAndReplanIn subroutine that leverages information reuse.

GrowAndReplanIn(w, Π) is implemented to reuse the search information available from the last window in the new, larger window. This reuse of part of the search tree from the previous iteration is a key idea exploited by X*.

This subroutine operates much like standard A*; it uses an open list, O, to hold the search frontier, and a closed list, C, to hold already expanded states, with states $s \in O$ expanded in the order of minimum f-value, $f(s)$, and this minimum state accessed by top(O). GrowAndReplanIn also has a state neighbor function, $N(s)$, which returns the set of collision-free neighbors of s. In addition, it also uses the unique concept of an "out-of-window" list, X, which stores states removed from O and intended to be expanded, but are outside of the current window boundary. These states are stored in X for use in the next search. Finally, GrowAndReplanIn reasons about the path between the successive window starts b_2 and b_1 along the path, π, and accesses the cost of π via $\|\pi\|$.

Figure 1 shows the three stages which make up X*'s GrowAndReplanIn procedure. The first stage transforms a search tree from b_1 to e_1 in w_1 into a search tree from b_1 to e_1 in w_2. The second stage transforms the search tree from b_1 to e_1 in w_2 into a search tree from b_2 to e_1 in w_2. The third stage transforms the search tree from a from b_2 to e_1 in w_2 into a search tree from b_2 to e_2 in w_2.

Stage 1 This stage, presented in Algorithm 2, reintroduces states from X to O, as they would have naturally been expanded by an A* search from b_1 to e_1 if run in w_2. It then runs A* until the smallest f-value in O is greater than or equal to the f-value of e_1, $f(e_1)$. It also repairs the f-values of all the states in the C, such that they are optimal in w_2 for e_1.

Precondition 1 *Valid A* search tree formed by O and C from b_1 to e_1 in w_1.*

Postcondition 1 *Valid A* search tree formed by O and C from b_1 to e_1 in w_2.*

Proof Sketch 1 *We know that, from Precondition 1, all states expanded in the search of w_1 were expanded because their f-value is less than or equal to the f-value of e_1. Thus, if a state was prevented from being expanded by the boundary of w_1, and thus placed in X, in w_2 that state should be reconsidered for expansion. Furthermore, it is possible that there are states in C which, due to the constraints imposed by w_1, have suboptimal f-values in w_2. To handle this,* A*SearchUntil *will re-expand a state already in C if that state is removed from the top of O with a lower f-value (Algorithm 5, Postcondition 4, Line 5). Thus, after Line 4, we know that we have a valid A* search tree formed by O and C from b_1 to e_1 in w_2, and thus Postcondition 1 is met.*

Algorithm 2 Stage 1

```
1: procedure Stage1
2:     $O \leftarrow O \cup X$
3:     $X \leftarrow \emptyset$
4:     A*SearchUntil($O, C, X, w_2, f(e_1)$)
```

Stage 2 This stage, presented in Algorithm 3, moves the start of the search tree from b_1 to b_2 in w_2. The optimal path from b_2 to b_1, π, is known via the individually optimal plans, and thus both its cost, $\|\pi\|$, and the states along the path, $s \in \pi$, are known.

The f-value of all states in O and C are increased by $\|\pi\|$, (Line 2), and states along the path π are expanded (Line 3).

Precondition 2 *Valid A* search tree formed by O and C from b_1 to e_1 in w_2.*

Postcondition 2 *Valid A* search tree formed by O and C from b_2 to e_1 in w_2.*

Proof Sketch 2 *We know from Precondition 2 that we have a valid search tree from b_1 to e_1; if we connect b_2 to b_1 via an optimal path, we can insert the existing tree into the new tree. We know by construction π is optimal, as the individual plans were optimal, and collision-free, as they are not part of the initial collision.*

Thus, extending the f-value of all states $s \in O \cup C$ by $\|\pi\|$ serves as the minimum cost to reach these states from b_2 through b_1 to s, as per the definition of π and Precondition 2, and thus serves as an upperbound on the cost to go from b_2 to s. The algorithm then expands, using A*SearchUntil, *all states with an f-value lower than the extended f-value of e_1, re-expanding any states in the closed list with a lower f-value than that which was closed (Algorithm 5, Postcondition 4, Line 5). Thus, a valid A* search tree formed by O and C from b_2 to e_1 in w_2, and Postcondition 2 is met.*

Algorithm 3 Stage 2

```
1: procedure Stage2
2:     for all $s \in O, C$ do $f(s) \leftarrow f(s) + \|\pi\|$
3:     for all $s \in \pi$ do
4:         $C \leftarrow C \cup \{s\}$
5:         $O \leftarrow O \cup N(s)$
6:     A*SearchUntil($O, C, X, w_2, f(e_1) + \|\pi\|$)
```

Stage 3 This stage, presented in Algorithm 4, moves the goal of the search tree from e_1 to e_2. This is done by recalculating the heuristics for all openlist states and continuing the execution of A* with the existing O and C.

Precondition 3 *Valid A* search tree formed by O and C from b_2 to e_1 in w_2.*

Postcondition 3 *Valid A* search tree formed by O and C from b_2 to e_2 in w_2.*

Proof Sketch 3 *We know from Precondition 3 that the search tree spans to e_1. In addition, we know that there exists a path from e_1 to e_2 by construction of these goals. Thus, by running standard A* with O and C towards e_2 (Lines 3 - 13), we know from the properties of A* that the result is a valid A* search tree formed by O and C from b_2 to e_2 in w_2, thus satisfying Postcondition 3.*

Algorithm 4 Stage 3

```
1: procedure STAGE3
2:     for all $s \in O, C$ do $h(s) \leftarrow H(s, e_2)$
3:     while $O \neq \emptyset$ do
4:         $s \leftarrow \text{top}(O)$
5:         if $s = e_2$ then return UNWINDPATH($C, e_2, b_2$)
6:         $O \leftarrow O \setminus \{s\}$
7:         if $s \in C$ then continue
8:         if $s \notin w$ then
9:             $X \leftarrow X \cup \{s\}$
10:            continue
11:        $C \leftarrow C \cup \{s\}$
12:        $O \leftarrow O \cup N(s)$
13:    return NOPATH
```

A* Search Until presented in Algorithm 5, is a modified version of A* where an inconsistent heuristic is used; it expands a state $s \in O$ if that state $s \notin C$, *or* if $s \in C$, but s was placed in C with a higher f-value than s's current f-value. In addition, rather than halting when the goal is found, A*SEARCHUNTIL halts when the f-value of the state at the top of the openlist has an equal or greater value than the given limit f_{max}. This ensures that, assuming a valid A* search tree was given via O and C, the final search tree will have a frontier as if standard A* was run from b inside w_2.

Precondition 4

1. $\forall s \in C : \exists$ *a path to the start from* s *through the standard A* unwind.*
2. $\forall s \in C, \forall n \in N(s) : (n \in C \vee n \in O) \wedge g(n) = g(s) + c(s, n)$.
3. $f_{max} \geq \text{top}(O)$

Postcondition 4 *Standard A* search tree from* b *to a frontier of cost greater than or equal to* f_{max}.

Proof Sketch 4 *We know from Precondition 2 that all states* $s \in C \cup O$ *have been properly expanded, but the order of expansion may be different than that of Standard A*. However, this algorithm allows for state re-expansion, as shown on Line 5, to ensure that if a suboptimal g-value associated with a state* s *was previously expanded and placed in* C*,* s *can be re-expanded with the optimal g-value of* s*. As per Line 2, we know that the algorithm halts when all states with an f-value less than or equal to* f_{max} *have been processed, which means these states have had their optimal g-value, and thus optimal f-value, assigned, and as per Precondition 3 this means that all of the states in* C *have been properly expanded. Thus, we have formed a standard A* search tree from* b *to a frontier of cost greater than or equal to* f_{max}.

Algorithm 5 A* Search Until

```
1: procedure A*SearchUntil($O, C, X, w, f_{max}$)
2:     while $f(\text{top}(O)) < f_{max}$ do
3:         $s \leftarrow \text{top}(O)$
4:         $O \leftarrow O \setminus \{s\}$
5:         if $\exists s' \in C : s = s' \wedge f(s) \geq f(s')$ then
6:             continue
7:         if $s \notin w$ then
8:             $X \leftarrow X \cup \{s\}$
9:             continue
10:        $C \leftarrow C \cup \{s\}$
11:        $O \leftarrow O \cup N(s)$
```

3.4 Lazy Neighbor Expansion

A* with an exact heuristic is known to perform the minimal number of state expansions possible in order to find an optimal solution [7]; however, on graphs with a high average degree, the runtime of real-world A* implementations is usually dominated by managing neighbors and managing the heap data structure backing the openlist, O [22,8]. Approaches such as Partial Expansion A* [22] and Enhanced Partial Expansion A* [8] seek to remedy this issue by only partially adding neighbors to O, and performing book-keeping to add more of the neighbors later if needed.

Our approach, show in Algorithm 6, introduces a generator $G := (s, n_{\text{min}}, l)$ which is responsible for maintaining knowledge of the neighbors of s, including the unexpanded joint neighbor state with the lowest f-value, n_{min}, as well as the ability to generate the next of such a state if it exists, stored in the set l.

Algorithm 6 Generator Functions

```
1: function MakeGenerator($s$)
2:     return $\left(s, \text{argmin}_{n \in N(G.s)} : f(n), \emptyset\right)$
3: procedure GetNextState($G$)
4:     if $G.n_{\text{min}} = \text{argmin}_{n \in N(G.s)} : f(n)$ then
5:         $G.l \leftarrow N(G.s)$
6:     $G.l \leftarrow G.l \setminus \{G.n_{\text{min}}\}$
7:     $G.n_{\text{min}} \leftarrow \text{top}(G.l)$
8: function IsExhausted($G$)
9:     return $G.n_{\text{min}} \neq \text{argmin}_{n \in N(G.s)} : f(n) \wedge G.l = \emptyset$
```

A generator state expansion introduces a single generator, G, to O, ordered using the value of $f(G.n_{\text{min}})$, which serves as a lower bound on the f-value of all states in $N(G.s)$. Where b is the branching factor of the joint graph and n is the number of expanded and un-closed states, a generator expansion only requires a single heap percolation rather than b heap percolations, and it reduces $\|O\|$ by a factor of b, making the generator addition to an openlist of $O(\log n)$, rather than $O(b^2 \log bn)$ for an eager neighbor expansion. The generator also avoids initially storing the full $N(s)$ set, saving the storage space in the common case where $G.s$ is expanded but none of its neighbors are expanded. G is removed from O when $G.n_{\min}$ has been expanded and $\textsc{IsExhausted}(G)$ is true.

Furthermore, Algorithm 4 Line 2 requires a full reordering of O's heap. The generator model allows for the reordering of each generator (an $O(b)$ operation) to be deferred until use, and drops the cost of an O reordering from $O(bn)$ for eager to $O(n)$ for lazy, thereby saving the reorder cost for generators that are not queried after the reorder.

We compare Lazy X* to Eager X* to demonstrate the performance improvement of the Lazy Neighbor Expansion optimizations for a single interaction. In order demonstrate this performance, we run X* on the scenario Circular Swap:

Circular Swap (CS) Agents are placed on the edge of a circle and each agent attempts to swap places with the agent diametrically across from it. This scenario represents the worst case for all multiagent planning algorithms, as it requires a high degree of interaction for all agents.

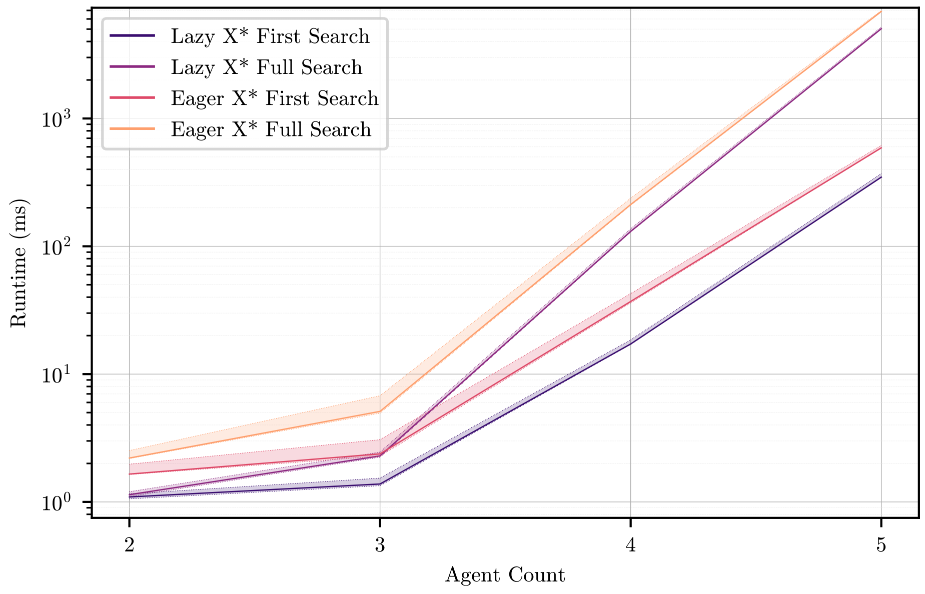

Fig. 2: Runtimes of Eager X* and Lazy X* on CS; 95% confidence intervals over 100 trials.

Figure 2 shows X* run on CS with a starting window radius of 2 and a heuristic inflation of 1.0. The results show Lazy X* produces somewhere between a 2× and 8× performance improvement over Eager X*. Due to the significant performance improvement provided by Lazy Neighbor Expansion, all further experimentation is performed with Lazy Neighbor Expansion.

4 Experimental Results

We evaluate X* in 3 ways: 1) we explore the behavior of X* across its configuration settings; 2) we compare how X* scales with the number of agents involved in a single interaction versus several baseline and state-of-the-art algorithms; 3) we present the performance of X* vs several state-of-the-art optimal planners for several realistic scenarios.

4.1 Characterizing X*

Selecting Parameters

Heuristic Inflation In this work, X* assumes that the given heuristic is admissible and consistent, as these are required for X* to find optimal paths. However, a standard technique to speed up A* is to trade speed for optimality by inflating the search heuristic by an $\epsilon > 1$, which causes A* to produce a path with a cost that is within ϵ of the optimal path cost. X* also supports this technique to allow for bounded sub-optimal planning.

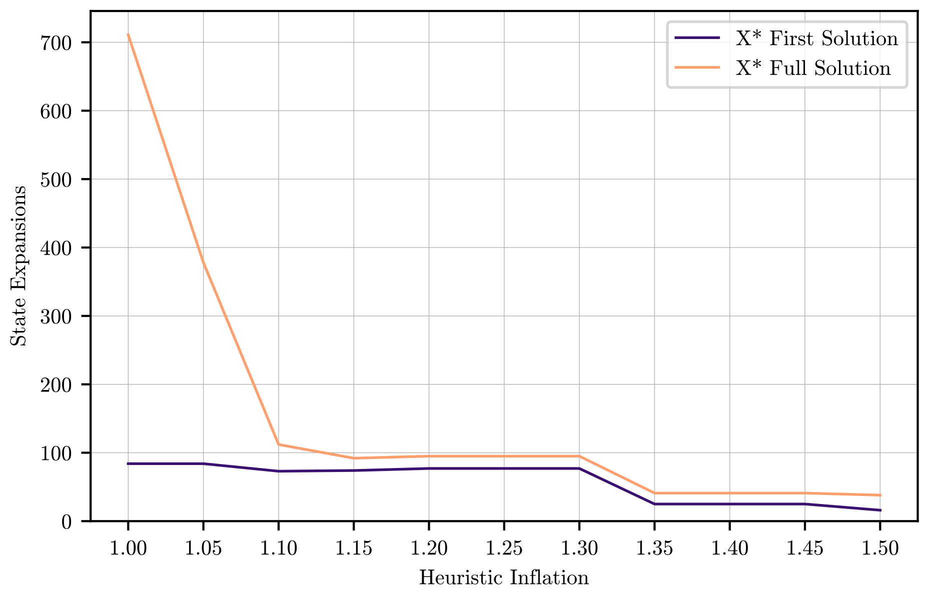

Fig. 3: X* state expansions vs heuristic inflation

Figure 3 shows X* run on CS with a starting window radius of 2 with 4 agents. This curve shows that the best solution quality vs plan time tradeoff is when inflation is approximately 1.1.

Initial Window Sizing Initial window selection determines how much of the joint space X* will search on its first run. Selecting too large of a window requires X* to search farther than necessary to generate a valid first solution, while selecting too small of a window requires X* to find a path in an unnecessarily or impossibly constrained environment.

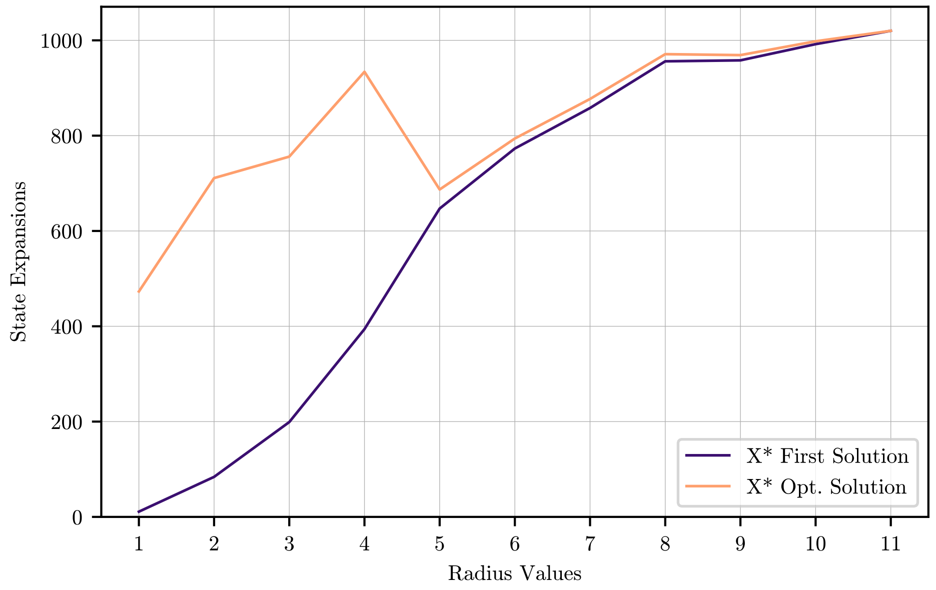

Fig. 4: X* state expansions vs initial radius

Figure 4 presents X* run on CS with 4 agents and a heuristic inflation of 1.0. This figure shows that finding a very small window that still encapsulates the collision will significantly decrease both the first search and full search overhead. Another interesting aspect of this result is that the peak in X* Full Solution expansions at radius of 4 represents a lack of preference towards two optimal solution homotopic classes, whereas the radius of 3 and 5 represent a preference induced by Algorithm 5 Line 2.

Minimizing Planning Dimensionality X* works by exploiting the sparse nature of multiagent interactions, so it is important that X* maintains low joint plan dimensionality when possible. To demonstrate this, we run X* on the scenario Non-interacting Groups:

Non-interacting Groups (NG) Agents are placed into groups of two, 10 grid steps from each other, and attempt to swap places. Multiple of these groups are geometrically separated from each other so that they do not interact. This scenario represents a case where the dimensionality of the interactions can be kept significantly below the dimensionality of full planning problem.

We ran X* on NG with 6 agents, an initial window radius of 2, and a heuristic inflation of 1.0. X* was successfully able to keep the dimensionality of these interactions in the space down to the two agents in each interaction. For the first solution in each interaction, X* required 5 expansions, and for the full solution in each interaction, X* required 25 total expansions

Quality and Computation Time vs Iteration Tradeoff As X* is an anytime algorithm, characterizing the quality vs iteration and quality vs time tradeoff is important to employing metareasoning [20]. To characterize this, we run X* on the scenario CS with 5 agents, an initial window radius of 2, and a heuristic inflation of 1.0. X* was able to successfully find an optimal solution in the first iteration, at a cost of 3 steps more than the individually optimal solution, and was able to prove it found an optimal solution in 5 iterations. Figure 5 presents the percentage of computation time each iteration consumes, averaged over 20 trials. This figure demonstrates that for a single interaction, X* is able to a first solution very quickly (in approximately 1% of the total computation time) and successive solutions come in at approximately consistent time intervals.

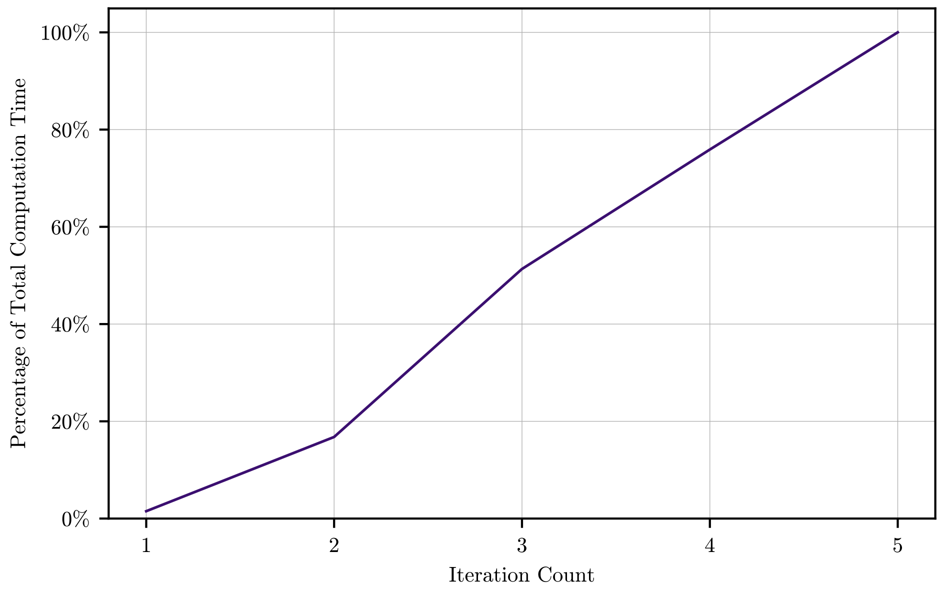

Fig. 5: Percentage of total computation time vs X* iteration

4.2 X* Scalability and Comparison Showcase

To demonstrate how well X* scales relative to baseline algorithms such as NWA* and full joint planning A*, we measured the number of state expansions as we scaled the number of agents while running the CS scenario.

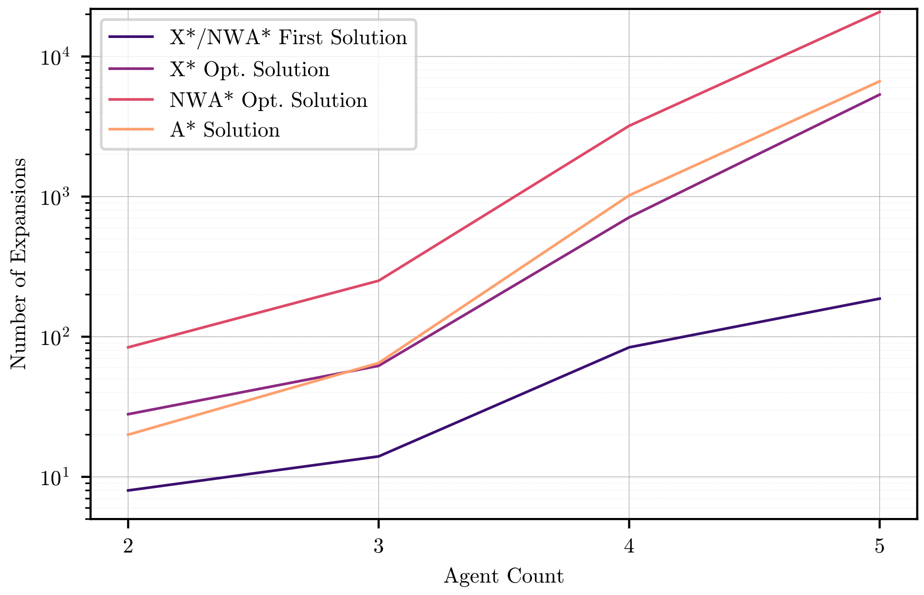

Fig. 6: Expansion comparison between X* and baseline algorithms

Figure 6 presents X*, NWA*, and A* run on CS with a heuristic inflation of 1.0 and a starting window radius of 2. As shown, NWA* and X* are consistently able to generate a first solution in approximately an order of magnitude fewer state expansions than A*. While NWA* ends up performing approximately half an order of magnitude more state expansions than A* to get an optimal solution, X* is able to perform roughly the same or fewer state expansions compared to A* to get an optimal solution. X* is able to outperform A* due to favorable expansion ordering as a result of Algorithm 5 Line 2.

Figure 7 presents a comparison between X*, M*, and CBS run on CS with a heuristic inflation of 1.0 and a starting window radius of 2. Due to the varying dimensionality of M*'s state expansions and the individual space planning of CBS, there is no meaningful comparison between expansion counts for these algorithms and that of X*. Instead, we use a "Normalized Runtime", presented as a 95% confidence interval over 100 instances of the wall clock runtime of each algorithm implementation divided by the wall clock runtime of an A* search for each agent in individual space in each implementation. This metric is more meaningful than raw wall clock time because it factors out different levels of

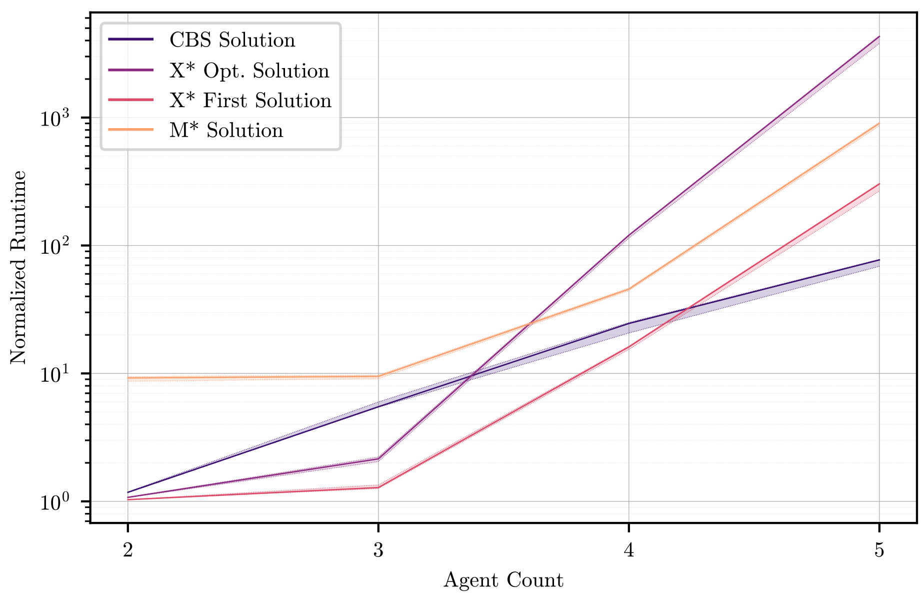

Fig. 7: Normalized runtime of X* and state-of-the-art with number of agents in a single interaction; 95% confidence intervals over 100 trials.

implementation optimization as well as different expenses of common operations, such as the cost of doing a single individual space occupancy check. To mitigate language differences such as garbage collection, all implementations used were written in C++, where the implementation of X* was produced by the authors of this paper, the implementation of M* was provided by the algorithm's authors[1], and the implementation of CBS was provided by a third party open source project[2]. All runtimes were measured on a dedicated computer with an i7 CPU, Turbo Boost disabled, and 32GB of RAM.

4.3 Realistic Scenarios

We compare X* against Operator Decomposition (OD) M* and CBS for two realistic randomized scenarios where optimal solutions are desirable, but anytime properties are a boon.

Multiple Robots Around Building (BUILD) Multiple agents are placed in random configurations in the hallways of the synthetic floor plan shown in Figure 8 with random unique start and goal locations. The same set of random scenario configurations were used for each planner to ensure fair comparisons. This is intended to be representative of a real-world configuration of delivery robots

[1] M* Source Code URL: `https://github.com/gswagner/mstar_public`
[2] CBS Source Code URL: `https://github.com/whoenig/libMultiRobotPlanning`

moving about the building while constantly having to replan due to the dynamic environment.

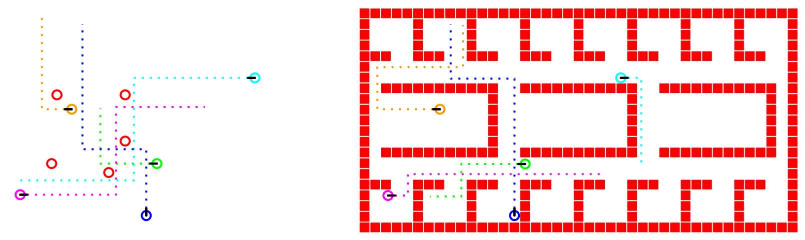

Fig. 8: RRSO and BUILD scenarios with example solutions. All red items are treated as static obstacles.

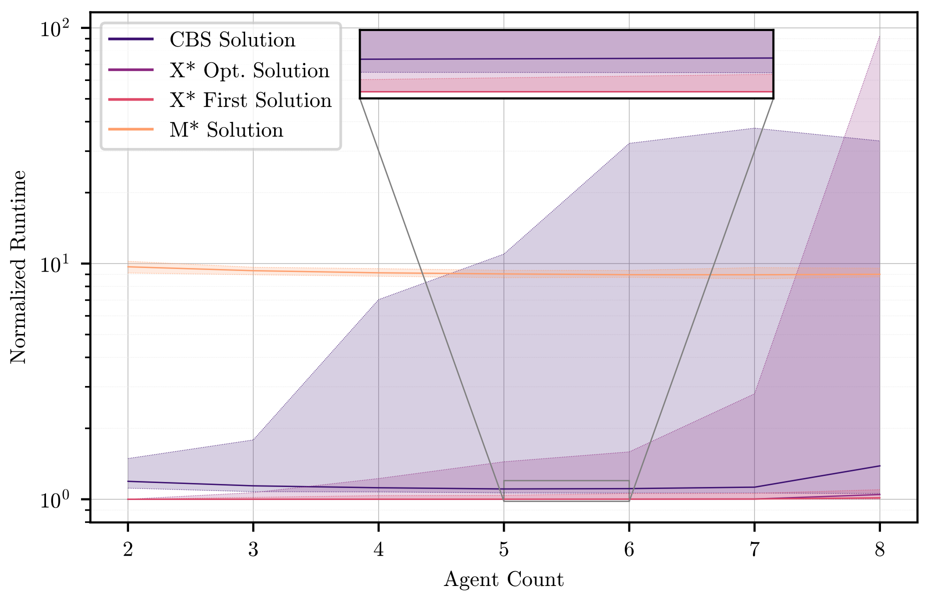

Fig. 9: Normalized runtime of X* and state-of-the-art in BUILD scenario; 95% confidence intervals over 100 trials.

Figure 9 presents BUILD run with an initial window radius of 2 and a heuristic inflation of 1.0. This figure shows that X* is consistently able to generate fast first solutions for 5 or fewer agents. While the confidence intervals of X*'s first solution and CBS overlap for 6+ agents, X*'s first solution mean is still below the bottom of CBS's interval. In addition, CBS has a high upper bound on its solution generation runtime while X*'s first solution interval is very tight, meaning CBS is less reliably able to quickly generate a solution than X*. In addition, this graph shows that while there is a very high upper bound in the speed of X*'s optimal plan generation, it has the lowest mean runtime of any of the optimal plans. Overall, this experiment demonstrates X*'s ability to quickly generate a first solution in very small amounts of time, faster than or more reliably than the state-of-the-art for multiagent planners, and it demonstrates X*'s ability to generate a full optimal solution in an amount of time competitive with the existing state-of-the-art.

Random Robot Soccer With Obstacles (RRSO) Agents are placed in random configurations in a 5000mm × 4000mm section of the RoboCup Small Size League field with random unique start and goal locations and a static opposing team, an example of which is shown in Figure 8. The same set of random scenario configurations were used for each planner to ensure fair comparisons. This scenario is intended to be representative of a real-world configuration of a RoboCup SSL field during a game, such as shortly after an indirect free kick.

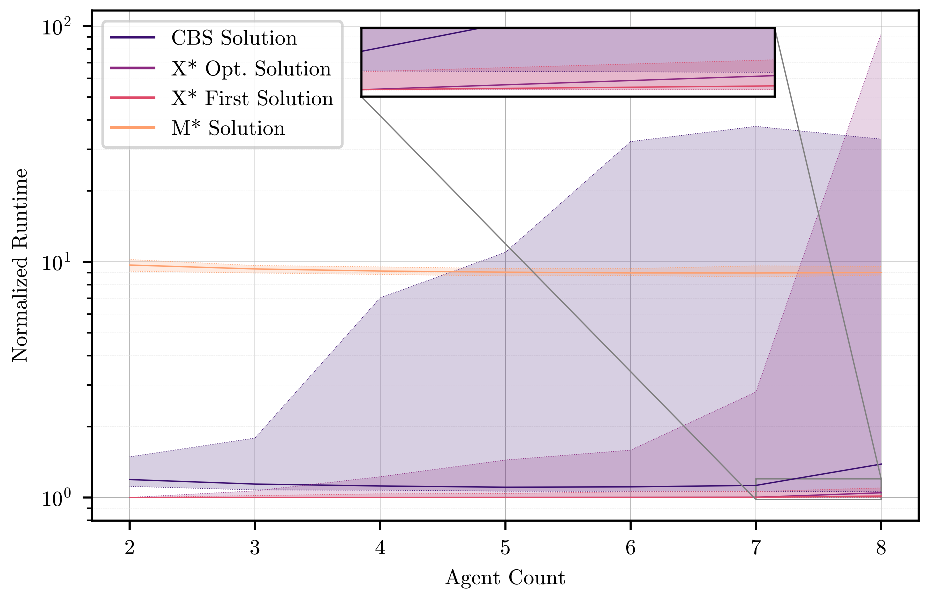

Fig. 10: Normalized runtime of X* and state-of-the-art in RRSO scenario; 95% confidence intervals over 100 trials.

Figure 10 presents RRSO run with an initial window radius of 2 and a heuristic inflation of 1.0. This figure shows that X* is consistently able to generate fast first solutions for 7 or fewer agents. While the confidence intervals of X*'s first solution and CBS overlap for 8 agents, X*'s first solution mean is still below the bottom of CBS's interval. Once again, CBS has a high upper bound on its solution generation runtime while X*'s first solution interval is very tight, meaning CBS is less reliably able to quickly generate a solution than X*. In addition, this graph shows that once again while there is a very high upper bound in the speed of X*'s optimal plan generation, it has the lowest mean runtime of any of the optimal plans, and even sits below the lower bound of CBS. Overall, this experiment demonstrates X*'s ability to quickly generate a first solution in very small amounts of time, faster than or more reliably than the state-of-the-art for multiagent planners, and it demonstrates X*'s ability to generate a full optimal solution in a competitive amount of time with the existing state-of-the-art.

In addition, the BUILD and RRSO scenarios demonstrate that a major shortcoming of M* is its inability to quickly generate solutions in domains with low dimension interactions. As stated in Section 2, this is due to the fact that M*'s runtime in these domains is dominated by the individual agent plan generation, and thus M* spends most of its computation time calculating the individually optimal policies of all agents, an expensive computation, whereas X* runs simple A* for all agents, a less expensive computation.

5 Conclusion and Future Work

In this work we present a technique to perform anytime multiagent planning by iteratively repairing individual paths projected into joint space. Specifically, we present the concept of geometric search space bounding, a novel transformation to allow for the grafting of these trees in larger search spaces, and a novel neighbor representation to allow for lower memory usage and lazy heap updates to speed these transformations.

At its core, X* uses A* to perform window searches; with the exception of Lazy Neighbor Expansion, the A* implementation has no advanced features. Despite this, X* is still able to achieve better performance than the state-of-the-art for first plan generation and competitive performance for full plan generation. Due to the fact that the features of WAMPF and the specific techniques of X* are mostly orthogonal to other MPP solving techniques, we believe that using more advanced A*-like planners and potentially non A*-like planners to perform the window search while still allowing for window reuse will produce a new generation of anytime multiagent planners that combine the strengths of X* and the advanced planner.

Furthermore, we believe that the techniques we present are not limited simply to path planning; we believe that if we can define a window in the search space for an arbitrary graph planning problem and the problem has a mutex relation for two or more sub-searches, our window concept plus tree search preservation technique applied to existing solutions would produce a fast anytime planner.

References

1. Aine, S., Chakrabarti, P.P., Kumar, R.: AWA*-a Window Constrained Anytime Heuristic Search Algorithm. In: Proceedings of the 20th International Joint Conference on Artifical Intelligence. pp. 2250–2255. IJCAI'07 (2007)
2. Aine, S., Likhachev, M.: 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)
3. Aine, S., Likhachev, M.: Truncated Incremental Search. Artificial Intelligence **234**(C), 49–77 (May 2016)
4. Barer, M., Sharon, G., Stern, R., Felner, A.: 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)
5. Cohen, L., Greco, M., Ma, H., Hernández, C., Felner, A., Kumar, T.K.S., Koenig, S.: Anytime focal search with applications. In: IJCAI (2018)
6. Crosby, M., Jonsson, A., Rovatsos, M.: A Single-agent Approach to Multiagent Planning. In: Proceedings of the Twenty-first European Conference on Artificial Intelligence. pp. 237–242. ECAI'14, IOS Press, Amsterdam, The Netherlands, The Netherlands (2014)
7. Dechter, R., Pearl, J.: Generalized best-first search strategies and the optimality of A*. In: Journal of the Association for Computing Machinery. vol. 32, pp. 505–536 (1985)
8. Goldenberg, M., Felner, A., Stern, R., Sharon, G., Sturtevant, N., Holte, R.C., Schaeffer, J.: Enhanced Partial Expansion A*. J. Artif. Int. Res. **50**(1), 141–187 (May 2014)
9. Hopcroft, J., Schwartz, J., Sharir, M.: On the Complexity of Motion Planning for Multiple Independent Objects; PSPACE - Hardness of the "Warehouseman's Problem". In: The International Journal of Robotics Research. pp. 76–88 (1984)
10. Karaman, S., Frazzoli, E.: Sampling-based algorithms for optimal motion planning. In: International Journal of Robotics Research. vol. 30, pp. 846–894 (2011)
11. Kavraki, L.E., Svestka, P., Latombe, J.C., Overmars, M.H.: Probabilistic roadmaps for path planning in high-dimensional configuration spaces. In: IEEE Transactions on Robotics and Automation. vol. 12, pp. 566–580. IEEE (1996)
12. Koenig, S., Likhachev, M.: D* Lite. In: Proceedings of the AAAI Conference of Artificial Intelligence. pp. 476–483. AAAI (2002)
13. Kushleyev, A., Likhachev, M.: Time-bounded lattice for efficient planning in dynamic environments. 2009 IEEE International Conference on Robotics and Automation pp. 1662–1668 (2009)
14. Likhachev, M., Gordon, G., Thurn, S.: ARA*: Anytime A* with Provable Bounds on Sub-Optimality. In: Advances in Neural Information Processing Systems 16: Proceedings of the 2003 Conference (2003)
15. Narayanan, V., Phillips, M., Likhachev, M.: Anytime Safe Interval Path Planning for dynamic environments. 2012 IEEE/RSJ International Conference on Intelligent Robots and Systems pp. 4708–4715 (2012)
16. Ryan, M.R.K.: Exploiting Subgraph Structure in Multi-Robot Path Planning. J. Artif. Intell. Res. **31**, 497–542 (2008)
17. Sharon, G., Stern, R., Felner, A., Sturtevant, N.R.: Conflict-based search for optimal multi-agent pathfinding. Artificial Intelligence **219**, 40 – 66 (2015)
18. Silver, D.: Cooperative Pathfinding. In: Proceedings of the First AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment. pp. 117–122. AIIDE'05, AAAI Press (2010)

19. Sun, X., Yeoh, W., Koenig, S.: Generalized Fringe-Retrieving A*: faster moving target search on state lattices. In: AAMAS (2010)
20. Svegliato, J., Zilberstein, S.: 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)
21. Wagner, G.: Subdimensional Expansion: A Framework for Computationally Tractable Multirobot Path Planning. Ph.D. thesis, The Robotics Institute Carnegie Mellon University (2015)
22. Yoshizumi, T., Miura, T., Ishida, T.: A* with Partial Expansion for Large Branching Factor Problems. In: Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence. pp. 923–929. AAAI Press (2000)
23. Zhou, R., Hansen, E.A.: Multiple sequence alignment using A*. In: Proceedings of the AAAI Conference of Artificial Intelligence (2002)