X*: ANYTIME MULTIAGENT PLANNING WITH BOUNDED SEARCH

Kyle Vedder

Completion Date:

December 12, 2018

Approved By:

Joydeep Biswas, Committee Chair, College of Information and Computer Sciences

---

Shlomo Zilberstein, Committee Member, College of Information and Computer Sciences

---

Abstract

Title: **X*: ANYTIME MULTIAGENT PLANNING WITH BOUNDED SEARCH**

Author: **Kyle Vedder**

Thesis/Project type: **Manuscript/Conference Paper**

Approved by: **Joydeep Biswas, Committee Chair, College of Information and Computer Sciences**

Approved by: **Shlomo Zilberstein, Committee Member, College of Information and Computer Sciences**

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 the global optimal plan. Building upon these two insights, we introduce a sparse, anytime variant of the A* planner, which we call X* (Expanding A*). X* implements two novel techniques to reduce the computational cost compared to joint A*: 1) it preserves the partial X* search trees and priority queues between iterations of window growth; and 2) it defers explicit joint state enumeration until necessary. By preserving the search tree, X* significantly out-performs joint A* and a naïve window-growing A* algorithm. By deferring explicit joint state enumeration, X* reduces the number of priority queue operations by several orders of magnitude compared to a joint A* planner. We present empirical results from several domains, showing that X* outperforms existing state-of-the-art joint planners for sparse anytime multi-agent planning with optimality convergence.

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 [1] [2] and repairing them when environments change or obstacles move [3]. 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 rectangular agents is PSPACE-hard [4].

There are multiple ways the characterize the quality of an MPP solution given a set of agent plans Π depending on the domain. For instance, there's the total cost, characterized as

$$\sum_{\pi \in \Pi} \|\pi\|$$

which is useful in domains where you are charged for total energy consumption, such as gas usage in your various delivery vehicles. There's other metrics, such as the makespan, characterized as

$$\max(\{\|\pi\| \mid \pi \in \Pi\})$$

which is useful in domains where you are trying to minimize maximum delay to goal arrival, such as in a warehouse delivering time sensitive goods. While selecting a particular metric to quantify cost does not fundamentally change any of the planning algorithms, this works chooses to focus on minimizing the total cost of all plans.

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.

In this work we focus on domains with an arbitrary number of agents but with sparse agent interaction. 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.

This work presents an anytime solver for the multiagent planning problem that is competitive with or faster than the existing state-of-the-art by leveraging bounded search, novel replanning techniques, and novel neighbor management to speed plan generation and enhancement.

In particular, we present three key contributions in this work. First, we show that by exploiting the sparsity of these interactions we can quickly solve each interaction locally, allowing for fast first solution generation. Second, we present a transformation to embed the search tree of a given prior search into the search tree of a relaxed search while preserving optimality, thereby speeding the search in the relaxed window. Third, we present lazy neighbor expansion, which reduces memory overhead and improves runtime performance compared to its eager counterpart. Note that, while the second and third contributions only apply to A*-like planners, the first contribution is applicable to a variety of search techniques. With these three key contributions, we demonstrate a significant performance advantage over state-of-the-art solvers for time to first solution while still providing competitive optimal solution generation time.

2 Significance

The Multiagent Planning Problem is applicable to a wide array of areas, from individual agent based goal attainment such as package delivery or warehouse management, to cooperative based domains such as formation planners or robot soccer teams. In many of these domains, there exists a tradeoff between planning time and quality of solution. For example, in a warehouse management domain, there's a tradeoff between time spent planning for agents and time spent executing the

plans. For instance, if a few more milliseconds spent planning would produce several seconds in savings for plan execution, further time spent planning is worth its cost. However, in another case, if several seconds more planning time is needed to reduce the execution of a plan by only a few milliseconds, further time spent planning is not worth its cost. As a result, we see that the ability to make a conscious decision between planning time and result quality is an important aspect of real world multiagent systems, and this is what anytime planning provides.

3 Background

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

3.1 Search Reuse

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 [3], propagates changes in the environment back up the search tree, only modifying states g-values as needed. This enables the planner to often times perform only a small fraction of the total number of expansions otherwise needed if replanning from scratch. Other examples of algorithms that employ reuse are from the predator-prey domain, such as Generalized Fringe Retrieving A* (GFRA*) [5]. GFRA* works by pruning off sections of the search tree from the previous search that involves paths not taken by the predator, and then running A* from that point towards the new prey with the existing search tree, thereby saving the states inside the pruned tree from needing to be re-expanded.

3.2 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 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 [6], via depth of search such as Hierarchical Cooperative A* (HCA*) [7], or via restricted cost propagation such as Truncated D* Lite (TD*) [8].

The time-bounded lattice is intended for domains where obstacle positions are parameterized by a time t. Initially, the space is discretized on t, but after a set time horizon, reasoning about obstacles moving is no longer of much value but is still computationally very expensive. Thus, after that time horizon, the time-bounded lattice makes it such that positions are no longer considered to be parameterized by t, reducing the planning space dimensionality and thus reducing planning cost.

Hierarchical Cooperative A* is an MPP solver that uses a reservation table to track prior used positions of serially planned paths. Optionally, HCA* also allows for bounding of search depth; that is, after a fixed number of steps, HCA* halts planning for agents. This is a significant cost savings while still providing reasonable quality, as with most domains in practice there is rarely a need to modify a large swath of the early stage of the plan when planning for the later plan.

Truncated D* Lite is a repair algorithm based in D* Lite that uses a non-zero threshold to determine when to stop back propagation of inconsistency in states. This is done in an effort to avoid back propagation of small amounts of inconsistency that ultimately will have minimal effect on the resulting solution. Varying the bound varies the quality guarantee that D* Lite provides, but in many cases even a small bound can translate into a large computational cost savings.

3.3 Anytime Planning

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 [9]. 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 [10]. 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, such as Anytime Repairing A* (ARA*) [11] and Anytime Window A* (AWA*) [12].

ARA* works by searching with a heavily inflated heuristic, and, as time permits, iteratively re-propagates information through the tree about tighter and tighter heuristic bounds. This allows for much of the tree structure to remain unchanged. AWA* works by using a sliding window concept. It starts by performing depth first search and gradually proceeds towards A* by incrementing the window size.

There exists other, non A*-like anytime path planners which also leverage this concept of reuse, such as RRT* [1], 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 [13] [14].

3.4 Multiagent Planning

Prior work on planners designed to solve the MPP 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 [7] and planning in general [15]. 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] [17].

M* is a state-of-the-art A*-like global solver for optimal and ϵ-suboptimal MPPs [17]. It operates by first finding an optimal policy for the full individual configuration space of each agent, and then combines 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. A shortcoming of M* is that the full individual policy calculation is expensive, making it slow for realtime planning.

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) [18] and its ϵ-suboptimal counterpart [19]. 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.

4 Methodology

In this work we merge the above techniques to present Expanding A* (X*), an anytime MPP solver that leverages search bounding and search reuse for fast first and successive plan generation. X* operates by first planning for each agent independently, and then identifying interacting groups of agents from these individual plans. Next, for each interacting group, X* projects the individual plans into the joint planning space of the group, and constructs a joint space *window*, an artificial geometric bound, around the point of interaction. X* then proceeds to repair the collision in this window. While there exists remaining time in the planning time budget, X* then iteratively grows the window and improves the existing plan.

Algorithm 1 presents the general anytime sparse interaction algorithm for multi-agent path planning called Anytime Multiagent Path Planner (AMPP). AMPP first computes individual plans for each agent (Line 2). Next, it computes the sets of agents that have individual plans that interfere with each other (Line 3). For all such sets, AMPP computes the smallest window in the joint state space over the interacting agents (Line 4), such that the window completely encompasses all joint states where the agents interact with each other. For all windows, AMPP attempts to plan a joint path through the window (Line 5). Finally, until the available computation time is exhausted (Line 6), AMPP iteratively increases the size of the window, computing joint plans within the window at each step (Line 7). The iterative nature of AMPP allows the joint path search on Line 7

to reuse part of the search tree from the previous iteration – this a key idea exploited by Expanding A*.

A naïve alternative to Expanding A* is an algorithm where, in Line 7, instead of reusing the search tree during the iterative growth of the window, it re-computes the joint plan from scratch for every window. As we shall show empirically, this results in significantly higher computational cost. We refer to this alternative algorithm as the Naïve Window A* (NWA*).

AMPP is also optimal in the limit under the assumptions that searches within each window are optimal and colliding windows will be merged together in Line 7. This is due to the fact that with sufficient iterations, all windows will grow large enough that they will not restrict the search tree for interacting agents, thereby allowing the search to form a valid A* search tree from the initial start to the initial goal, thus finding an optimal solution. Note that for X* this status is effectively checked by seeing if the out of window list X, defined in Section 4.2, is empty.

There are two key subroutines to AMPP: COMPUTEWINDOWS, representing Line 4, which computes the interacting sets of agents and their corresponding windows, and GROWANDREPLAN, representing Line 7, which iteratively grows the search tree from a smaller window to a larger window. We first present how COMPUTEWINDOWS operates.

Algorithm 1 Anytime Multiagent Path Planner

```
1: procedure AMPP
2:     Plan independently for all agents
3:     S ← interacting set(s) of agents
4:     W ← {w | w is the smallest fit window for s ∈ S}
5:     ∀w ∈ W, plan jointly in w
6:     while more time available do
7:         ∀w ∈ W, grow w and replan jointly
```

4.1 Compute Windows

A window W is an artificial constraint placed on the search space of the interacting set of agents in order to allow for a focused search to quickly find a joint collision free path, while still encompassing the entire collision. For our experimentation, we model a window as a cube centered at

the interaction center and defined via a radius, i.e. an L_∞ norm, from the center. It possesses a start b and goal e, each sitting on a face of the window and defined to be in the joint space of the interacting agents, but the position of each agent is on that agent's individually defined path as it enters and exits the window in individual space. In addition, for an arbitrary window W_1, it has a successor, W_2, where $W_1 \subset W_2$.

The result of COMPUTEWINDOWS is a set of windows which collectively encompass all of the joint space collisions of the individually planned paths from Line 2. Given each of these windows, AMPP plans jointly for the interacting set of agents in the window (Line 5) before using the GROWANDREPLAN subroutine, which we present next.

4.2 Grow And Replan

Like in standard A*, GROWANDREPLAN 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). GROWANDREPLAN 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, GROWANDREPLAN reasons about the path between the successive window starts b_2 and b_1 along the individually planned path, π, and accesses the cost of this path via $\|\pi\|$.

Figure 1 shows the three stages of the GROWANDREPLAN procedure. GROWANDREPLAN's 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 ← O ∪ X
3:     X ← ∅
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 know both its cost, $\|\pi\|$, and the states along the path, $s \in \pi$, are known.

Using $\|\pi\|$, the f-value of all the states in O and C are increased (Line 2), and states along the path 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 ∈ O, C do h(s) ← H(s, e_2)
3:      while O ≠ ∅ do
4:          s ← top(O)
5:          if s = e_2 then return UNWINDPATH(C, e_2, b_2)
6:          O ← O \ {s}
7:          if s ∈ C then continue
8:          if s ∉ W then
9:              X ← X ∪ {s}
10:             continue
11:         C ← C ∪ {s}
12:         O ← O ∪ N(s)
13:     return NOPATH
```

A* Search Until

A*SEARCHUNTIL, presented in Algorithm 5, is a modified version of A* with an inconsistent heuristic; 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(top(O)) < f_max do
3:          s ← top(O)
4:          O ← O \ {s}
5:          if ∃s' ∈ C : s = s' ∧ f(s) ≥ f(s') then
6:              continue
7:          if s ∉ W then
8:              X ← X ∪ {s}
9:              continue
10:         C ← C ∪ {s}
11:         O ← O ∪ N(s)
```

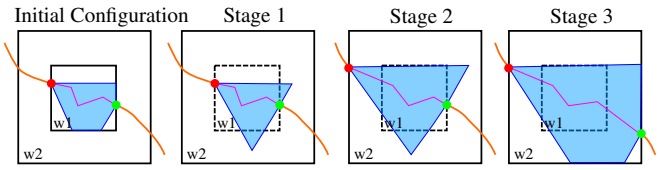

Figure 1: Stages of `GrowAndReplan`. How X* reuses search information between windows.

4.3 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 [20]; 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. Approaches such as Partial Expansion A* [21] and Enhanced Partial Expansion A* [22] seek to remedy this issue by only partially adding neighbors to O, and performing book-keeping to add the more of the neighbors later if needed.

Our approach introduces a neighbor generator $G := (s, n_{\text{min}}, l)$ which is responsible for maintaining knowledge 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.

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 $G.n_{\text{min}}$ has been expanded and 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.

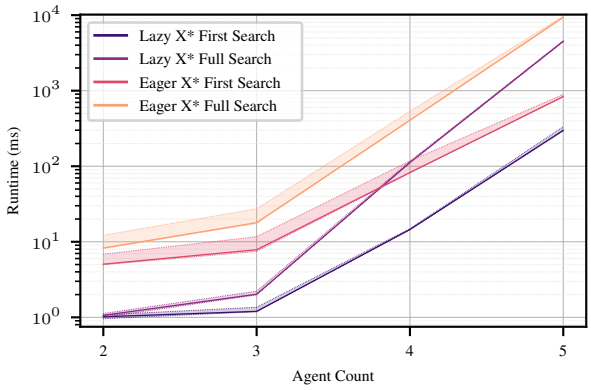

Figure 2: Runtimes of Eager X* and Lazy X*; 95% confidence intervals over 100 trials.

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 $2\times$ and $8\times$ performance improvement over Eager X*. Due to the significant performance improvement provided by Lazy Neighbor evaluation, all future experimentation will be performed with Lazy Neighbor evaluation.

5 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 the state-of-the-art for several realistic scenarios.

5.1 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.

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

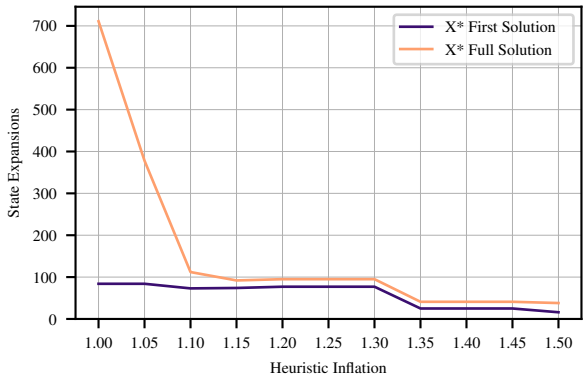

Figure 3: X* state expansions vs heuristic inflation

Initial Window Sizing Initial window selection determines how much of the search 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.

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 the tie-breaking of Algorithm 5 Line 2.

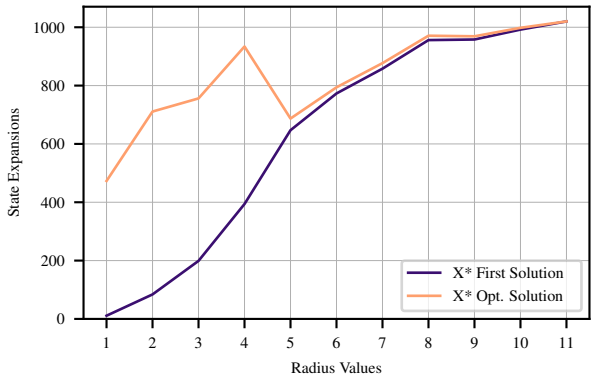

Figure 4: X* state expansions vs initial window radius

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 [9]. 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. 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.

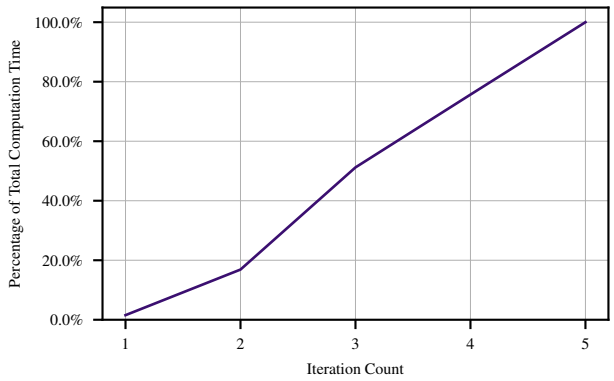

Figure 5: Percentage of total computation time vs X* iteration

5.2 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.

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 tiebreaking as a result of Algorithm 5 Line 2.

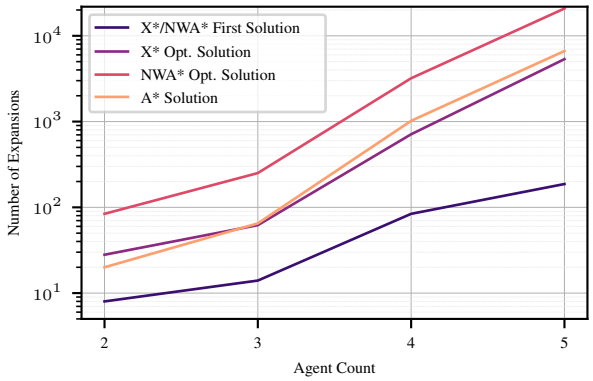

Figure 6: Expansion comparison between X* and baseline algorihms

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. This metric is more meaningful than raw wall clock time because it factors out different levels of 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.

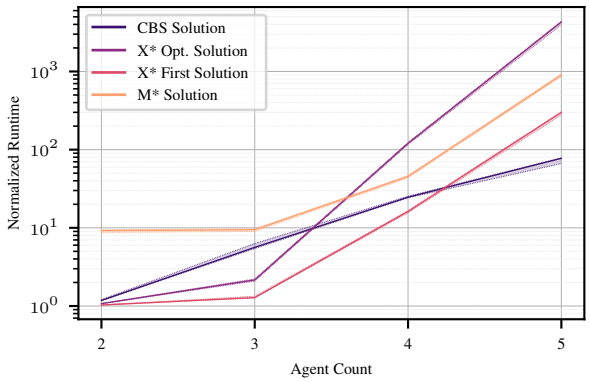

Figure 7: Normalized runtime of X* and state-of-the-art with number of agents in a single interaction; 95% confidence intervals over 100 trials.

Figure 7 shows that X* is significantly faster than the existing state-of-the-art for generating first and even final solutions for lower dimensional interactions, but, while the first solution generation outperforms M*, it does not scale as well as CBS to higher dimensions.

5.3 Realistic Scenarios

We compare X* against M* and CBS for two realistic randomized scenarios where anytime planners could want to be deployed.

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

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

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

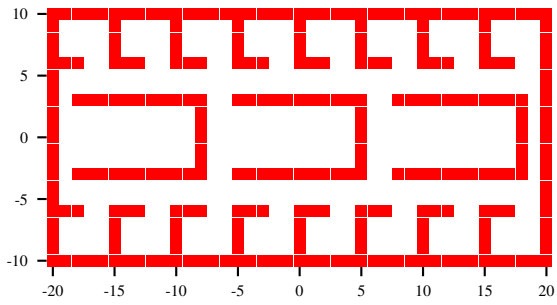

Figure 8: BUILD scenario floorplan

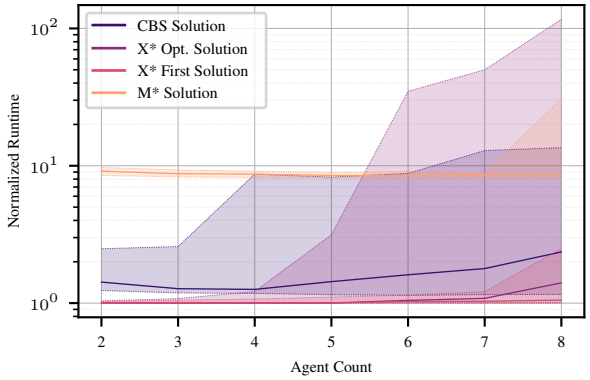

Figure 9: Normalized runtime of X* and state-of-the-art in BUILD scenario; 95% confidence intervals over 100 trials.

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 often 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 $5000\text{mm} \times 4000\text{mm}$ section of the RoboCup Small Size League field with random unique start and goal locations and a static opposing team. 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.

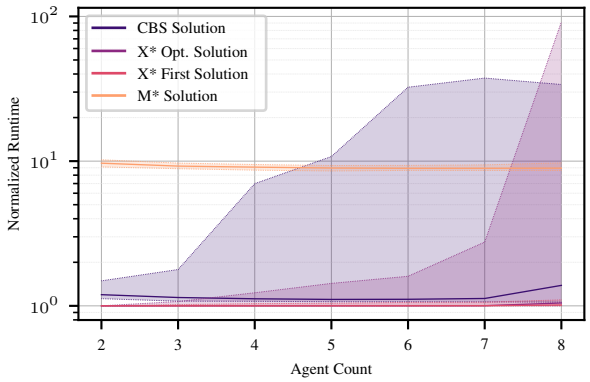

Figure 10: Normalized runtime of X* and state-of-the-art in RRSO scenario; 95% confidence intervals over 100 trials.

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 often 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 3, 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.

6 Conclusion

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 evaluation, 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 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 combining the strengths of X* and the strengths of 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] S. Karaman and E. Frazzoli. "Sampling-based algorithms for optimal motion planning". In: *International Journal of Robotics Research*. Vol. 30. 2011, pp. 846–894.

[2] L. E. Kavraki et al. "Probabilistic roadmaps for path planning in high-dimensional configuration spaces". In: *IEEE Transactions on Robotics and Automation*. Vol. 12. IEEE. 1996, pp. 566–580.

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

[4] J.E. Hopcroft, J.T. Schwartz, and 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.

[5] Xiaoxun Sun, William Yeoh, and Sven Koenig. "Generalized Fringe-Retrieving A*: faster moving target search on state lattices". In: *AAMAS*. 2010.

[6] Aleksandr Kushleyev and Maxim Likhachev. "Time-bounded lattice for efficient planning in dynamic environments". In: *2009 IEEE International Conference on Robotics and Automation* (2009), pp. 1662–1668.

[7] David Silver. “Cooperative Pathfinding”. In: *Proceedings of the First AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*. AIIDE’05. Marina del Rey, California: AAAI Press, 2010, pp. 117–122.

[8] Sandip Aine and Maxim Likhachev. “Truncated Incremental Search”. In: *Artificial Intelligence* 234.C (May 2016), pp. 49–77. ISSN: 0004-3702.

[9] Justin Svegliato and Shlomo 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.

[10] R. Zhou and E. A. Hansen. “Multiple sequence alignment using A*”. In: *Proceedings of the AAAI Conference of Artificial Intelligence*. 2002.

[11] M. Likhachev, G. Gordon, and 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.

[12] Sandip Aine, P. P. Chakrabarti, and Rajeev Kumar. “AWA*-a Window Constrained Anytime Heuristic Search Algorithm”. In: *Proceedings of the 20th International Joint Conference on Artifical Intelligence*. IJCAI’07. Hyderabad, India, 2007, pp. 2250–2255.

[13] Sandip Aine and Maxim 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.

[14] Venkatraman Narayanan, Mike Phillips, and Maxim Likhachev. “Anytime Safe Interval Path Planning for dynamic environments”. In: *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems* (2012), pp. 4708–4715.

[15] Matthew Crosby, Anders Jonsson, and Michael Rovatsos. “A Single-agent Approach to Multiagent Planning”. In: *Proceedings of the Twenty-first European Conference on Artificial Intelligence*. ECAI’14. Prague, Czech Republic: IOS Press, 2014, pp. 237–242. ISBN: 978-1-61499-418-3.

[16] Malcolm Ross Kinsella Ryan. "Exploiting Subgraph Structure in Multi-Robot Path Planning". In: *J. Artif. Intell. Res.* 31 (2008), pp. 497–542.

[17] G. Wagner. "Subdimensional Expansion: A Framework for Computationally Tractable Multirobot Path Planning". PhD thesis. The Robotics Institute Carnegie Mellon University, 2015.

[18] Guni Sharon et al. "Conflict-based search for optimal multi-agent pathfinding". In: *Artificial Intelligence* 219 (2015), pp. 40–66. ISSN: 0004-3702.

[19] Max Barer et al. "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.

[20] R. Dechter and J. Pearl. "Generalized best-first search strategies and the optimality of A*". In: *Journal of the Association for Computing Machinery*. Vol. 32. 1985, pp. 505–536.

[21] Takayuki Yoshizumi, Teruhisa Miura, and Toru Ishida. "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*. AAAI Press, 2000, pp. 923–929. ISBN: 0-262-51112-6.

[22] Meir Goldenberg et al. "Enhanced Partial Expansion A*". In: *J. Artif. Int. Res.* 50.1 (May 2014), pp. 141–187. ISSN: 1076-9757.