Shard Systems: Scalable, Robust and Persistent Multi-Agent Path Finding with Performance Guarantees

Christopher Leet, Jiaoyang Li, Sven Koenig

USC Department of Computer Science
cjleet@usc.edu, jiaoyanl@usc.edu, skoenig@usc.edu

Abstract

Modern multi-agent robotic systems increasingly require scalable, robust and persistent Multi-Agent Path Finding (MAPF) with performance guarantees. While many MAPF solvers that provide some of these properties exist, none provides them all. To fill this need, we propose a new MAPF framework, the shard system. A shard system partitions the workspace into geographic regions, called shards, linked by a novel system of buffers. Agents are routed optimally within a shard by a local controller to local goals set by a global controller. The buffer system novelly allows shards to plan with perfect parallelism, providing *scalability*. A novel global controller algorithm can rapidly generate an inter-shard routing plan for thousands of agents while minimizing the traffic routed through any shard. A novel workspace partitioning algorithm produces shards small enough to replan rapidly. These innovations allow a shard system to adjust its routing plan in real time if an agent is delayed or assigned a new goal, enabling *robust*, *persistent* MAPF. A shard system's *local optimality* and *optimized inter-shard routing* bring the sum-of-costs of its solutions to single-shot MAPF problems to between 25% and 70% of optimal on a diversity of workspaces. Its scalability allows it to plan paths for thousands of agents in seconds. If any of their goals change or move actions fails, a shard system can replan in under a second.

Introduction

In the *robust, persistent Multi-Agent Path Finding* problem, we are given a set of k *agents* $\mathcal{A} := \{a_1, \ldots, a_k\}$ positioned at k unique vertices in a *workspace* $G := (V, E)$. At any time, a *user* may issue at *task* $(a_i, v_j) \in \mathcal{A} \times V$ directing an agent $a_i \in \mathcal{A}$ to visit a *goal* $v_j \in V$. Agents service tasks first-come-first-served. An agent with no outstanding tasks is said to be *idle*. Idle agents can be moved freely.

Time is discretized into timesteps. At each timestep, an agent a_i must either move to an adjacent vertex v_j using a *move action* $a_i.move(v_j)$ or wait at its current vertex $a_i.v$ using a *wait action*. Two agents *conflict* if they either occupy the same vertex or traverse the same edge on the same timestep. A move action may randomly fail. If an agent a_i's movement fails, any move action which would move an agent a_j onto the vertex a_i occupies $a_i.v$ also fails. This simulates agents being able to avoid collisions with stationary objects using simple proximity sensors. Let an agent a_i's *path* p_i be the sequence of move and wait agents the agent takes to visit each of its goals in turn. Let the cost of p_i be the number of actions $|p_i|$ it contains. The goal is to generate a conflict-free path for each agent such that these paths' *sum-of-costs* $\sum_i |p_i|$ is minimized. If an agent is assigned a new goal or experiences movement failure, its path must be updated in real-time (*i.e.,* a small fraction of a second).

One important special case of the robust, persistent MAPF problem is the *one-shot MAPF problem*, where each agent is initialized with a single goal, no further goals are assigned, and movement never fails.

An increasing number of real world autonomous systems need to solve instances of the robust, persistent MAPF problem with thousands of agents to operate. Ideally, moreover, these solutions should have close-to-optimal sum-of-costs. Warehouse operators, such as Amazon, use thousands of agents grouped into 800+ agent teams to retrieve items stored in large warehouse complexes (Simon 2019). Entertainment companies, such as Spaxels, produce aerial spectaculars featuring thousands of UAVs flying in concert (Schranz et al. 2020). Airport scheduling requires coordinating the movement of thousands of planes (Yu and LaValle 2016a). Unfortunately, while robust, persistent MAPF solvers exist, none can simultaneously scale to thousands of agents while producing solutions with close-to-optimal sum-of-costs. As such, the question:

Is it possible to build a robust, persistent MAPF solver which can produce solutions with close-to-optimal sum-of-costs for teams of thousands of agents?

is both open and of great practical relevance.

Shard System. We answer this question in the affirmative by developing a new MAPF framework, the *shard system*. Figure 1(a) shows the architecture of a shard system. A shard system partitions a workspace into subgraphs called *shards* $s_j \in S$ (green boxes) linked by short, unidirectional paths called *buffers* $b_k \in B$ (brown boxes). An agent in a shard is routed by the shard's *local controller*. An agent in a buffer is routed along the buffer's path to the buffer's head. Buffers and shards exchange agents using *transfer protocols*.

A *global controller* (yellow box) receives each task issued by a user (white box). The global controller executes

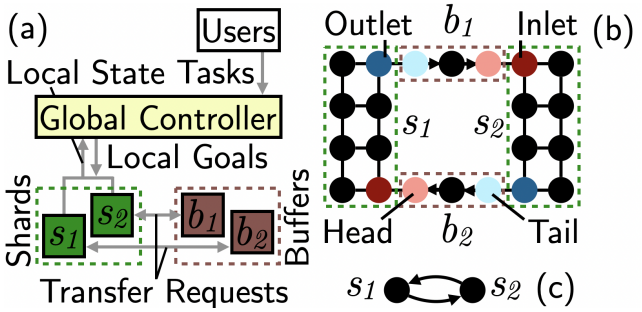

Figure 1: (a) The architecture of a shard system, (b) an example shard system and (c) this shard system's shard digraph.

an agent's task by sending each shard's local controller a *local goal* to route the agent to. By picking appropriate local goals, the global controller can have an agent passed from shard to shard until it reaches its goal.

For brevity, we will let the term *shard* refer to both a subgraph of G and the local controller managing that subgraph. We will thus speak of a shard containing agents but also of a shard routing agents. We will treat the term *buffer* similarly. Additionally, we will refer to an agent a_i's goal as a_i's *global goal* to differentiate it from a_i's *local goal*.

A shard system's shards and buffers collectively partition G, that is, any vertex $v \in V$ appears in exactly one shard or buffer. Let the set of vertices and edges that a shard contains be denoted $V(s_i) \subset V$ and $E(s_i) \subset E$ respectively. Each shard s_i is an induced subgraph of G, that is, there is an edge between two vertices $v_k, v_l \in V(s_i)$ in s_i iff $(v_k, v_l) \in E$. Fig. 1(b) depicts an example workspace partitioned into two shards (s_1 and s_2) and two buffers (b_1 and b_2). Each buffer's directionality is shown by depicting its edges as arcs.

A shard system's connectivity is captured by the *shard digraph* $G_S := (S, E_S)$, which contains: **(a)** a vertex for every shard $s_i \in S$ and **(b)** an arc (s_i, s_j) from shard s_i to shard s_j iff a buffer carries agents from s_i to s_j. A shard digraph must be strongly connected. Fig. 1(c) shows the example workspace's shard digraph.

Transfer Protocols. Let the shard that a buffer b_k receives agents from and sends agents to be termed b_k's *source shard* $b_k.src$ and *destination shard* $b_k.dst$ respectively. Let the first and last vertex in b_k's path be termed b_k's *tail* $b_k.tail$ and *head* $b_k.head$ respectively. A buffer's tail is connected to a unique *outlet* $u \in V(b_k.src)$ in its source shard such that $(u, b_k.head) \in E$. Similarly, a buffer's head is connected to a unique *inlet* $v \in V(b_k.dst)$ in its destination shard such that $(b_k.dst, v) \in E$. Figure 1(b) depicts b_1 and b_2's tails in light blue and heads in light red. Figure 1(b) depicts s_1 and s_2's outlets in dark blue and inlets in dark red.

If an agent a_i's local goal is an outlet, its shard s_j sends the buffer b_k connected to that outlet the *transfer request* "can I move a_i to your tail?" the timestep after a_i reaches this outlet. If b_k refuses a_i (because it is full), shard s_j routes a_i to a randomly chosen *waypoint* $waypt[a_i] \in V(s_j)$ and back, reducing congestion around b_k's outlet. Routing an agent to waypoint compromises sum-of-costs. To avoid this, we analyze buffer behavior with queueing theory, and allocate each buffer enough space to make it filling up unlikely.

When an agent a_i arrives at a buffer b_k's head $b_k.head$, b_k sends its destination shard $b_k.dst$ the transfer request "can I move a_i to your inlet?" If the shard refuses b_j (because the inlet is occupied), b_k tries again next timestep.

Performance Guarantees. A shard system's global controller and shards provide performance guarantees which help reduce its solutions' sum-of-costs. Let an agent a_i's *shard path* $sPath[a_i]$ on G_S be the sequence of shards that a_i passes through on its way to its global goal. Let a shard s_j's *utilization* U_j be the number of agents a_i whose shard paths contain s_j, that is $U_j := |\{a_i \in \mathcal{A} : s_j \in sPath[a_i]\}|$.

- *Guarantee 1.* An agent's shard path has minimal length.
- *Guarantee 2.* The maximum utilization of any shard $max(U_j : s_j \in S)$ is minimized .

These performance guarantees help decrease agent travel distance and the maximum congestion of any shard.

Let an agent a_i's *local path* $lPath_j[a_i]$ in a shard s_j be the sequence of actions that a_i takes to reach its current local goal or waypoint. Let the cost of agent a_i's local path be the number of actions $|lPath_j[a_i]|$ it contains.

- *Guarantee 3.* The sum-of-costs $\sum_i |lPath_j[a_i]|$ of the local paths assigned by a shard is bounded-suboptimal.

This performance guarantee ensures that the shard system's sum-of-costs is not increased by poor routing within shards.

Runtime. A shard system's buffers allow its shards to operate with perfect parallelism, greatly decreasing runtime. If one of the agents in a shard has its movement fail or local goal change, the shard must replan. Shards can replan in real-time because their subgraphs are small, limiting the maximum number of agents they can contain.

Empirical Performance. The shard system can solve the instances of the one-shot MAPF problem on a diversity of workspaces in seconds. Its performance guarantees allow it to find solutions with sum-of-costs between 25% and 70% of optimal. The shard system substantially outperforms existing state-of-the-art robust, persistent MAPF solvers. It can replan for a team of thousand of agents in a under a second if an agent's move action fails or global goal changes.

Limitations. The shard system has three key limitations. First, it is not complete. Since buffers are one way, connecting two shards in a tree graph with a buffer disconnects the workspace. Empirically, however, the shard system runs on all but one workspace (a tree graph) in the MAPF benchmark suite (Stern et al. 2019) (Section). Second, the shard system has no optimality bound. Third, our current implementation does not allow an agent's goal to be in a buffer.

Related Work

Single-Shot MAPF. The single-shot MAPF problem has been solved optimally by approaches such as Conflict Based Search (CBS) (Sharon et al. 2012) and reduction to Integer Linear Programming (ILP) (Yu and LaValle 2016b). CBS,

in particular, is well optimized; optimizations such as bypassing conflicts (Boyarski et al. 2015) and symmetry reasoning (Li et al. 2020) allow CBS to scale to many tens of agents. Ultimately, however, these approaches have failed to scale to hundreds of agents.

The need for scalability prompted the development of bounded-suboptimal single-shot MAPF solvers, such as Enhanced CBS (Barer et al. 2014). State-of-the-art search-based bounded-suboptimal MAPF solvers such as EECBS (Li, Ruml, and Koenig 2021) scale to a few hundred agents in dense workspaces (*i.e.,* workspaces with many obstacles and high agent to vertex ratio), and up to a thousand agents in open workspaces.

The difficulty of scaling even bounded-suboptimal single-shot MAPF solvers has led to interest in MAPF solvers which lack optimality guarantees but scale well. Examples include reactive MAPF solvers such as WHCA* (Silver 2005), rule-based MAPF solvers such as Push and Rotate (De Wilde, Ter Mors, and Witteveen 2014) and Parallel Push and Swap (Sajid, Luna, and Bekris 2012), anytime MAPF solvers such as X* (Vedder and Biswas 2019), and prioritized planning (Lozano-Pérez and Erdmann 1987). These MAPF solvers scale to a few thousand agents but tend to produce low quality solutions in dense workspaces.

Robust One-Shot MAPF Solvers. We term one-shot MAPF solvers which can handle move action failure *robust one-shot MAPF solvers* By prohibiting two agents from occupying the same vertex in a k timestep period, the k-delays framework (Atzmon et al. 2018) ensures that conflicts only arise if more than k of an agent's move actions fail but often increases sum-of-costs. A MAPF plan can be executed in an execution framework which resolves conflicts produced by delays, such as (Ma et al. 2017). Such frameworks, however, can produce low quality solutions in dense workspaces where one delay can cause many.

Robust, Persistent MAPF Solvers. Rule-based solvers, such as MAPP (Wang and Botea 2011), can provide robust, persistent MAPF but produce low-quality solutions in crowded environments. Search-based solvers such as (Švancara et al. 2019) exist which can incrementally update their previous solution. In dense environments, however, replanning typically affects many other agents, limiting these solvers' ability to scale by reusing prior computation. Execution frameworks, such as Action Dependency Graphs (Hönig et al. 2019), allow a MAPF solver to replan while its current plan is executed but are limited by the scalability of the underlying MAPF solver.

Partition-Based MAPF Solvers. We term MAPF solvers which partition the workspace geographically *partition-based MAPF solvers*. The `ros-mapf` framework (Pianpak et al. 2019) partitions the workspace along grid lines. Each partition is assigned an answer-set-programming-based controller. HMAPP (Zhang et al. 2021) partitions the workspace similarly. It uses a high-level MAPF solver for inter-partition routing. SDP (Wilt and Botea 2014) assigns each two-vertex-wide corridor in a workspace a specialized controller. Each subgraph connecting these corridors is assigned a WHCA*-based controller. The shard system introduces

Algorithm 1: Global Controller Timestep.

```
1:  function GlobalControllerTimestep
2:      for (a_i, v_k) ∈ Recv(users) do
3:          goalQ[a_i].enqueue(v_k)
4:      toRoute ← {a_i ∈ idle : |goalQ[a_i]| > 0}
5:      idle ← idle − toRoute
6:      if |toRoute| > 0 then
7:          for a_i ∈ toRoute do
8:              goalS[a_i] ← s_j s.t. goalQ[a_i][0] ∈ V_j
9:          sPath ← SPPlan(curS, goalS, G_S)
10:         for a_i ↦ (s_1, ..., s_n) ∈ sPath : a_i ∉ idle do
11:             lGoal[s_n][a_i] ← goalQ[a_i][0]
12:             for j ∈ [1, ... n − 1] do
13:                 lGoal[s_j][a_i] ← rndBuf(s_j, s_{j+1})
14:             for s_j ∈ V_s : s_j ∉ (s_1, ..., s_n) do
15:                 lGoal[s_j][a_i] ← None
16:     for obj ∈ S ∪ B do
17:         Send(obj, lGoal[obj] if obj ∈ S else None)
18:         objActive, objIdle ← Recv(obj)
19:         for a_i ∈ objActive ∪ objIdle do
20:             curS[a_i] ← obj if obj ∈ S else obj.dst
21:         for a_i ∈ objIdle do
22:             idle.add(a_i)
23:             goalQ[a_i].dequeue()
```

three key novelities which allow it to outscale these MAPF solvers while providing robust and persistent MAPF:

1. *Perfect Parallelism.* Prior partition-based MAPF solvers operate their partitions at least partially in serial. Buffers allow each a shard system's shards to operate with perfect parallelism, greatly improving scalability while limiting the effects of movement failure to a single shard.
2. *Novel Workspace Partitioning.* Prior partition-based MAPF solvers partition the workspace along grid lines or at bottlenecks. By leveraging the graph-partitioning literature, the shard system ensures that each shard is small (to allow for real-time replanning) and compact (to avoid their perimeters creating bottlenecks and detours).
3. *Novel Inter-Shard Routing.* The shard system's inter-shard routing algorithm can plan paths for thousands of agents in real time, outscaling prior partition based solvers' inter-partition routing algorithms.

The Shard System

We now describe in detail how a shard system's global controller, shards and buffers operate.

Global Controller. The global controller receives tasks submitted by users and executes these tasks by configuring each shard's local goals appropriately. At each timestep, the global controller (Algorithm 1):

Receives new tasks from its users (Lines 2-5). An agent a_i's goals are enqueued in its goal queue $goalQ[a_i]$. Let $idle$ be the set of idle agents. If an idle agent a_i is assigned a task, the global controller moves a_i from the set $idle$ to the set

$toRoute$ (Lines 4-5). If $toRoute$ isn't empty (Line 6), then the global controller:

Replans each agent's shard path (Lines 7-9). An agent a_i's shard path runs from its current shard $curS[a_i]$ to its goal shard $goalS[a_i]$, the shard containing the next goal in a_i's goal queue $goalQ[a_i]$. During replanning, any agent a_i in a buffer b_k is treated as if it had already arrived at the buffer's destination shard, $curS[a_i] = b_k.dst$.

Recomputes each shard's local goals (Lines 10-15). Shard s_j's local goal for an agent a_i, denoted as $lGoal[s_j][a_i]$ is:

1. a_i's global goal $goalQ[a_i][0]$, if s_j is a_i's goal shard, that is, $goalS[a_i] = s_j$ (Line 11).
2. an outlet to a random buffer leading to the next shard in a_i's path s_{j+1}, denoted as $rndBuf(s_j, s_{j+1})$, if s_j is any other shard on a_i's shard path (Lines 12-13).
3. $None$ if s_j is not on a_i's shard path (Lines 14-15).

Idle agents' local goals are left unchanged (keeping the agent at the last goal it was assigned). A shard s_j's local goal for agent a_i is initialized to $None$ if a_i doesn't start in s_j and $a_i.v$, a_i's current location, if a_i does.

Instructs each shard and buffer to execute a timestep (Lines 16-18). A shard s_j's instruction contains updated local goals $lGoal[s_j]$ (Line 17). The global controller waits for each shard and buffer to finish before proceeding (Line 18).

Updates its record of the system's state (Lines 19-23). Each shard and buffer sends the global controller its active and idle agents after finishing a timestep, allowing it to update each agent's current shard and the list of idle agents as well as remove completed goals from goal queues.

Shard. A shard s_j routes each agent $a_i \in A_j$ it controls to the agent's local goal $lGoal[a_i]$, possibly via a waypoint $waypt[a_i]$. Let an agent a_i's local path $lPath[a_i]$ in a shard s_j be represented as a list of vertices. A non-idle agent moves to the first vertex in its local path at each timestep and then removes this vertex. The first vertex in an agent a_i's local path is always adjacent to (indicating a move action) or identical to (indicating a wait action) the agent's current vertex $a_i.v$. An idle agent's path is empty.

Let a shard s_i's set of inlet vertices be denoted as V_i^{in}, outlet vertices as V_i^{out}, incoming buffers as B_i^{in} and outgoing buffers as B_i^{out}. Shard s_i's *inlet map* $f_i^{in} : V_i^{in} \to B_i^{in}$ and *outlet map* $f_i^{out} : V_i^{out} \to B_i^{out}$ map each inlet and outlet vertex respectively to the buffer it services. At each timestep, a shard (Algorithm 2):

Invalidates the local path of any agent with an updated local goal (Lines 2-4). If an agent's local goal has been updated, its local path is no longer valid. An agent's local path is flagged as invalid by setting it to $None$ (Line 3). A shard stores the local goals it was sent in the preceding timestep in $lGoalOld$ (Line 4). At initialization, $lGoalOld[a_i]$ is $a_i.v$ if $a_i \in A_j$ and $None$ otherwise.

Transfers agents to buffers (Lines 5-18). If an agent a_i's local goal is an outlet to a buffer b_k, shard s_j sends b_k a transfer request the timestep after a_i arrives (Lines 5-7). If b_k assents, a_i is moved to b_k's tail. If the move succeeds, a_i is moved from A_j to A_{b_k}, the set of agents in b_k (Lines 9-10). If the move fails, a_i's local path is invalidated (Line 12).

If b_k rejects a_i, b_k is full. To avoid congestion while b_k empties, s_j picks a vertex which is unlikely to be congested $uncongested(V_j)$, such as a vertex far away from any buffer, assigns it to a_i as a waypoint and sets $lPath[a_i]$ to $None$. (Lines 14-15). When a_i reaches the waypoint, its waypoint and local path are set to $None$ (Lines 16-18). If any agent $a_i \in A_j$ has an invalid local path, the shard:

Recomputes its local paths (Lines 19-20). An agent is routed to its waypoint $waypt[a_i]$ (if one exists) or its local goal $lGoal[a_i]$(otherwise).

Moves each agent along its local path (Lines 21-23). If an agent's movement fails, its local path is invalidated.

Responds to incoming transfer requests (Lines 24-26). The shard checks whether it has received a transfer request from any incoming buffer (Lines 24-25). It accepts a buffer's request iff its inlet vertex is unoccupied (Line 26).

Buffer. A buffer routes its agents from its tail to its head, where they wait to be transferred to its destination shard.

Handling Non-Idealized Conditions. Handling lossy, non-instantaneous communication and slow computation and movement requires adjustments to these algorithms. Messages are transmitted repeatedly in case of packet loss and stamped with the current timestep so that delayed, out-of-date messages from prior timesteps can be identified and

Algorithm 2: Shard Timestep for Shard s_j.

```
1:  function ShardTimestep(lGoal)
2:      for a_i ∈ A_j : lGoal[a_i] ≠ lGoalOld[a_i] do
3:          lPath[a_i] ← None
4:      lGoalOld ← lGoal
5:      for a_i ∈ A_j : a_i.v = lGoal[a_i] ∧ a_i.v ∈ V_j^out do
6:          b_k ← f_j^out(a_i.v)
7:          Send(b_k, "req")
8:          if Recv(b_k) = True then
9:              if a_i.mv(b_k.tail) then
10:                 Transfer(a_i, A_j, A_{b_k})
11:             else
12:                 lPath[a_i] ← None
13:         else
14:             waypt[a_i] ← uncongested(V_j)
15:             lPath[a_i] ← None
16:     for a_i ∈ A_j : a_i.v = waypt[a_i] do
17:         waypt[a_i] ← None
18:         lPath[a_i] ← None
19:     if ∃ a_i ∈ A_j : lPath[a_i] = None then
20:         lPath ← MAPF(A_j, waypt, lGoal)
21:     for a_i ∈ A_j : |lPath[a_i]| > 0 do
22:         if ¬ a_i.mv(lPath.dequeue()) then
23:             lPath[a_i] ← None
24:     for v_in ↦ b_k ∈ f_i^in do
25:         if Recv(b_k, "req") then
26:             Send(b_k, ¬∃ a_i ∈ A_j : a_i.v = v_in)
```

ignored. The global controller runs each shard and buffer's timestep in parallel. Each shard sends and awaits responses to its transfer requests in parallel. All agents are moved in parallel at the end of the timestep.

Shard Path Generation

The global controller minimizes the maximum utilization of any shard while minimizing each agent's shard path length.

Minimizing Maximum Utilization. Finding a routing plan which minimizes the maximum utilization of any shard is a variant of the multi-commodity flow problem. The set of agents C_{ij} in shard s_i with a global goal in shard s_j can be viewed as a commodity. $|C_{ij}|$ units of this commodity must be routed through the shard digraph (S, E_S) from s_i to s_j. Since agents are discrete, commodities have integer flows. We minimize the maximum flow through any shard $s_i \in S$. We do not limit the vertex or edge capacities.

This problem can be solved via integer linear programming (ILP). Let $\mathcal{C}$ be the set of all commodities C_{ij}, f_{ijkl} be commodity C_{ij}'s flow along the edge $(s_k, s_l) \in E_S$, and F_V be the maximum flow through any vertex. We minimize F_V subject to the following constraints:

(1) Each commodity C_{ij} in $\mathcal{C}$'s flow along each edge $(s_k, s_l) \in E_S$ is an integer between 0 and $|C_{ij}|$ (inclusive).

$$\forall\, C_{ij} \in \mathcal{C}, \forall\, (s_k, s_l) \in E_S,\ f_{ijkl} \in \{0, 1, \ldots, |C_{ij}|\}.$$

(2) Exactly $|C_{ij}|$ units of each commodity $C_{ij} \in \mathcal{C}$ are generated at s_i and consumed at s_j. Commodity C_{ij} is conserved at all other vertices.

$$\forall\, s_l \in S,\ \forall\, C_{ij} \in \mathcal{C},$$

$$\sum_{s_k:(s_k,s_l)\in E_S} f_{ijkl} \ - \sum_{s_k:(s_l,s_k)\in E_S} f_{ijlk} = \begin{cases} |C_{ij}| & l = j \\ -|C_{ij}| & l = i \\ 0 & otherwise \end{cases}.$$

(3) F_V is greater than or equal to the flow through any vertex. The flow through a vertex s_l is the sum of the flow entering s_l and the flow originating at s_l.

$$\forall\, s_l \in S,\ F_V \geq \sum_{s_k:(s_k,s_l)\in E_S} \ \sum_{C_{ij}\in\mathcal{C}} f_{ijkl} + \sum_{i:C_{li}\in\mathcal{C}} |C_{li}|.$$

The path that each unit of C_{ij}'s flow takes is assigned to a unique agent in C_{ij} whose current vertex is in shard s_i and whose goal is in shard s_j.

Minimizing Shard Path Length. We can minimize the maximum utilization of any shard when agents are constrained to minimum-length shard paths by adding additional constraints to the above formulation. Let P_{ij} be the set of all edges in (S, E_S) on a shortest path from s_i to s_j.

Theorem 0.1 *A shard path p on G_S from s_i to s_j is a shortest path iff each of p's edges is in P_{ij} .*

Theorem 0.1 is proved in (Cormen et al. 2009). By Theorem 0.1, confining commodity C_{ij} to edges in P_{ij} (by setting C_{ij}'s flow along any edge $(s_k, s_l) \notin P_{ij}$ to 0) forces the ILP solver to route C_{ij} along one or more shortest paths without disqualifying any shortest path from consideration. This constraint is incorporated into Constraint **(1)** as follows:

$$\forall\, C_{ij} \in \mathcal{C}, \forall\, (s_k, s_l) \in E_S,$$

$$f_{ijkl} \in \begin{cases} \{0, 1, \ldots, |C_{ij}|\} & (s_k, s_l) \in P_{ij} \\ \{0\} & (s_k, s_l) \notin P_{ij} \end{cases}.$$

Scaling Inter-Shard Routing. Our ILP formulation contains $O(|\mathcal{C}|(|E_S| + |S|))$ constraints. The runtime needed to solve ILPs scales exponentially in the number of constraints and thus runs impractically slowly for more than a couple hundred agents. We can approximately solve this multi-commodity flow problem by solving our formulation's Linear Programming relaxation and then generating an integer solution by randomized rounding. For any ϵ such that $0 < \epsilon < 1$, the maximum utilization of the solution provided by this procedure is $O(log(|E_S|/\epsilon)^{1/2})$ times the optimum (Raghavan and Tompson 1987). Even approximation takes time for teams of a couple thousand agents. At larger scales, we generate shard paths sequentially, picking the minimum-length shard path which increases the maximum shard utilization the least.

Workspace Partitioning

A shard system partitions its workspace into shards and buffers in three stages. *First*, we partition the workspace into shards. *Second*, we select an appropriate length for the shard system's buffers. *Third*, we generate buffers linking each shard to its neighbors.

Shard Generation. Workspace partitioning generates shards expected to contain the same, user-specified number n of agents. Let each vertex v_i in workspace G be assigned a weight $w(v_i)$ equal to the fraction of time it is expected to be occupied. (If the workspace's traffic cannot be predicted, each vertex's weight is set to the workspace's load factor α, which is the ratio of agents to vertices). Let a shard's weight $G_i := (V_i, E_i)$ be the sum of its vertices' weights $\sum_{v_i \in V_i} w(v_i)$. We would like to partition the workspace into $\lceil \sum_{v_i \in V} w(v_i)/n \rceil$ shards with the smallest range of weights possible. If there are multiple approximately equally good partitions, we select the one which cuts the minimum number of edges to make the shards as compact as possible.

Finding such a partition involves solving the graph partitioning problem, which is NP-hard (A. and R. 2004). Fortunately, this problem is well studied and approximate solvers exist (*e.g.,* (Karypis et al. 1998)).

Determining Buffer Length. We find the minimum length a buffer can be, given that its probability of overflow must be less than a user specified parameter $0 \leq \epsilon \leq 1$. A buffer's queue can be characterized by the parameters (Kendall 1953):

- *Inter-Arrival Time Distribution* (A). The distribution of time period between successive arrivals of agents.
- *Inter-Release Time Distribution* (B). The distribution of time periods between successive departures of agents.
- *Server Count* (X). The number of "servers" (here, shards) removing agents from a queue.
- *Capacity* (Y). The largest number of agents it can hold.

- *Queue Discipline* (Z). The order in which agents in the queue are removed.

A queue is typically denoted as an ordered list of these parameters, separated by slashes: $A/B/X/Y/Z$.

Exactly one shard removes agents from a buffer ($X = 1$). Agents are removed from a buffer first-come first-served (Z=$FCFS$). If a buffer has length n_{buf}, it can contain up to n_{buf} agents ($Y = n_{buf}$). A buffer's inter-arrival and inter-release time distributions are complex. To model these distributions, we assume that agents are evenly distributed across the workspace. If the workspace's load factor is α, a buffer has probability α of having an agent at its outlet vertex and probability $1-\alpha$ of not having an agent at its inlet vertex at each timestep. Its inter-arrival and inter-release time period lengths are thus geometrically distributed with rate parameters α and $1-\alpha$. In the limit, where a timestep becomes small compared to the overall runtime, the geometric distribution approaches the Poisson distribution. Over a sufficiently long runtime, a buffer's queue can thus be modeled as the queue $M(\alpha)/M(1-\alpha)/1/n_{buf}/FCFS$, where $M(\alpha)$ denotes the Poisson distribution with rate parameter α. The probability $Pr\{L_Q$=$i\}$ that this queue has queue length $L_Q = i$ at a given timestep is (Gross et al. 2008):

$$Pr\{L_Q = i\} = (\frac{\alpha}{1-\alpha})^i(1 - \frac{\alpha}{1-\alpha})[1 - (\frac{\alpha}{1-\alpha})^{n_{buf}+1}]^{-1}.$$

This queue's overflow probability is the probability the queue contains n_{buf} agents times the probability that another agent arrives $\alpha \cdot Pr\{L_Q = n_{buf}\}$. We pick the smallest value of n_{buf} whose overflow probability is less than ϵ.

While this model is crude, we find that it provides a reasonable first approximation. The model can be improved by recognizing that the space near a buffer's outlet and inlet is likely to have a higher effective load factor than the workspace as a whole and adjusting α accordingly. In our evaluations, we multiply α with a density factor of ν=1.33.

Buffer Placement. A buffer is generated by removing vertices from the shards it links. Buffer placement should leave a shard as compact as possible to avoid unnecessarily lengthening paths within the shard. We place buffers by generating a set of buffer templates of the appropriate length and enumerating the places each template can be placed. This process halts when a buffer placement is found that increases each shards' *perimeter*, the number of vertices in a shard adjacent to a vertex outside the shard, by less than a user specified value $bufQuality$. This approach, unfortunately, means that shard system generation is incomplete, because none of our templates may be a subgraph of a perverse workspace. Nonetheless, this approach was successful on a wide range of benchmark workspaces. Improving buffer placement is an important direction for future work.

Evaluation

Our evaluation studies the following questions:

1. **Coverage.** What is the space of workspaces which a shard system can solve MAPF instances on?
2. **Sum-of-costs.** How does the shard system's sum-of-costs compare to the state-of-the-art for one-shot MAPF?
3. **Scalability.** How does the shard system's scalability compare to the state-of-the-art for one-shot MAPF?
4. **Robustness.** How does the shard system's sum-of-costs compare to the state-of-the-art for one-shot MAPF with probablistic delays?
5. **Persistence.** How does the shard system's sum-of-costs compare to the state-of-the-art for persistent MAPF?
6. **Buffer Model.** How well do M/M/1/n_{buf}/FCFS queues model the shard system's buffers?

Workspace Type	**Description**
City Map	2D city scans
Dragon Age: Origins (DAO)	Video game maps
Dragon Age 2 (DA2)	Video game maps
Open	Empty workspaces
Open + Obstacles	Rooms with scattered 1×1 obstacles
Maze	Mazes of corridors
Room	Mazes of rooms
Warehouse	Regular grids of shelves

Table 1: The workspace types in the MAPF benchmark suite.

Workspace Name	**Workspace Type**	**Agents**
Small Workspaces (100-1,000 Agents)		
maze-32-32-2	Maze	99
random-32-32-10	Open + Obstacles	115
empty-32-32	Open	128
room-64-64-16	Room	455
ht_chantry	DA2	932
Large Workspaces (1,000-12,500 Agents)		
warehouse-10-20-10-2-2	Warehouse	1,222
lak303d	DAO	1,848
Boston_0_256	City Map	5,971
orz900d	DAO	12,075

Table 2: Each benchmark workspace's name, workspace type, and number of agents at a load factor of 0.125.

Implementation. The shard system is implemented in Python 3.9 and evaluated on a custom-built simulator. The shard system uses the Python multi-processing library for concurrency and the GLOP (Google 2021) integer linear programming solver to generate inter-shard routing plans. Shards route agents using the EECBS MAPF solver (Li, Ruml, and Koenig 2021). Workspace partitioning uses the hMETIS (Karypis et al. 1998) graph partitioning tool.

Benchmark Workspaces. Each evaluation (except for coverage) was run on a set of 9 workspaces taken from the MAPF benchmark suite (Stern et al. 2019). The MAPF benchmark suite contains 8 types of workspace (Table 1). Our set of workspaces contains every type of workspace.

The evaluations were run at a load factor of $\alpha = 0.125$. At this load factor, the smallest workspace in the MAPF benchmark suite contains 8 agents, and the largest 12,075 agents. The number of agents that our benchmark workspaces contain spans this range.

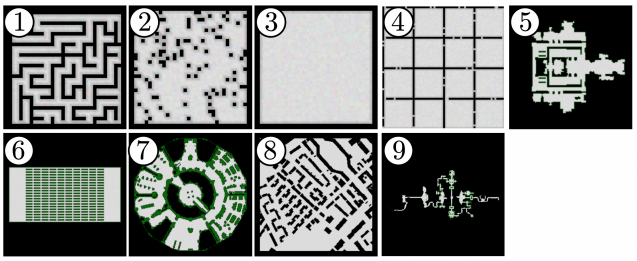

Figure 2: Thumbnails of: ① maze-32-32-2, ② random-32-32-10, ③ empty-32-32, ④ room-64-64-16, ⑤ ht_chantry, ⑥ warehouse-10-20-10-2-2, ⑦ lak303d, ⑧ Boston_0_256, and ⑨ orz900d.

City Map		**Open + Obstacles**	
Berlin_1_256	✓	random-32-32-10	✓
Boston_0_256	✓	random-32-32-20	✓
Paris_1_256	✓	random-64-64-10	✓
Dragon Age: Origins		random-64-64-20	✓
brc202d	✓	**Maze**	
den312d	✓	maze-32-32-2	✓
den520d	✓	maze-32-32-4	✓
lak303d	✓	maze-128-128-1	✗
orz900d	✓	maze-128-128-2	✓
ost003d	✓	maze-128-128-10	✓
Dragon Age 2		**Room**	
ht_chantry	✓	room-32-32-4	✓
ht_mansion_n	✓	room-64-64-8	✓
lt_gallowstemplar_n	✓	room-64-64-16	✓
w_woundedcoast	✓	**Warehouse**	
Open		warehouse-10-20-10-2-1	✓
empty-8-8	✓	warehouse-10-20-10-2-2	✓
empty-16-16	✓	warehouse-20-40-10-2-1	✓
empty-32-32	✓	warehouse-20-40-10-2-2	✓
empty-48-48	✓		

Table 3: The workspaces in the MAPF benchmark suite a shard system can and can't be applied to.

Each benchmark workspace's name, workspace type and number of agents for $\alpha = 0.125$ is shown in Table 2. Each benchmark workspace's thumbnail is shown in Figure 2.

Experimental Hardware. All evaluations were run on a Amazon EC2 c6a.48xlarge E2 instance with 96 CPU cores, 196 vCPU cores and 318 GiB of RAM.

Coverage

We evaluate the shard system's coverage by investigating whether there were any workspaces in the MAPF benchmark suite which could not be partitioned into a shard system with a strongly connected shard graph. As Table 3 shows, a valid layout could be found for every workspace except maze-128-128-1. Workspace maze-128-128-1 is a tree of one-vertex-wide corridors. Since all buffers are one-way, placing any buffer in this workspace prevents agents from moving from the buffer's destination shard to its source shard.

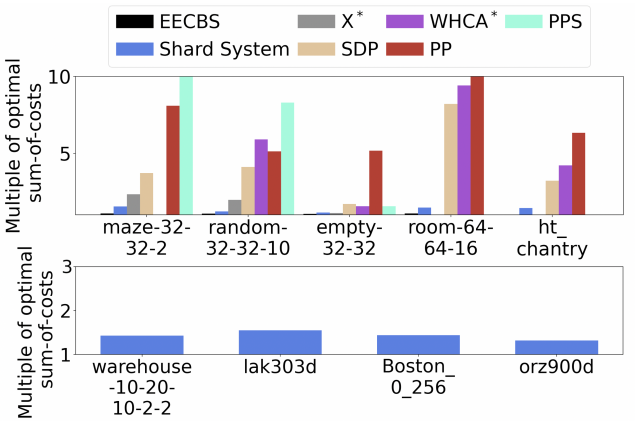

Figure 3: Comparing the shard system's sum-of-costs to that of 6 MAPF solvers.

Sum-of-Costs

We compared the shard system's sum-of-costs on the one-shot MAPF problem to that of 6 benchmark MAPF solvers:

1. *EECBS* (Li, Ruml, and Koenig 2021). A bounded sub-optimal conflict-based search MAPF solver with state-of-the-art extensions that improve its efficiency. It is run with a sub-optimality factor of $\omega = 1.1$.
2. *X** (Vedder and Biswas 2019). An anytime MAPF solver.
3. *WHCA** (Silver 2005). A reactive MAPF solver.
4. *SDP* (Wilt and Botea 2014). A MAPF solver which partitions a workspace into open regions and two vertex corridors. Open regions are assigned a WHCA*-based controller. Corridors are assigned a specialized controller.
5. *Parallel Push and Swap (PPS)* (Sajid, Luna, and Bekris 2012). A distributed rule-based solver.
6. *Prioritized Planning (PP)* (Lozano-Pérez and Erdmann 1987). A priority-based MAPF solver.

These six MAPF solvers were selected to represent the six classes of one-shot MAPF solver listed in Section .

Methodology. Ten one-shot MAPF instances were generated for each benchmark workspace. Each instance had a load factor of $\alpha \sim 0.125$. The agents were assigned randomly selected start and goal vertices from all vertices not part of a buffer. The runtime of all MAPF solvers was limited to 20 seconds. Let the cost of an agent a_i's path p_i be the number $|p_i|$ of actions it contains. We measure the mean sum-of-costs $\sum_i |p_i|$ that each solver achieved on each workspace's MAPF instances.

Results. Figure 3 (top) shows each MAPF solver's sum-of-costs on the small workspaces. Each measurement is presented as a multiple of the optimal. If a MAPF solver did not terminate on any MAPF instance, no measurement is given.

EECBS's mean sum-of-costs, as expected, was 108% to 110% of the optimal. X^*'s mean sum-of-costs was between 120% and 220% of the optimal. Neither solver terminated on any MAPF instance with more than 455 agents. The size of a large MAPF instance's space of possible solutions makes it

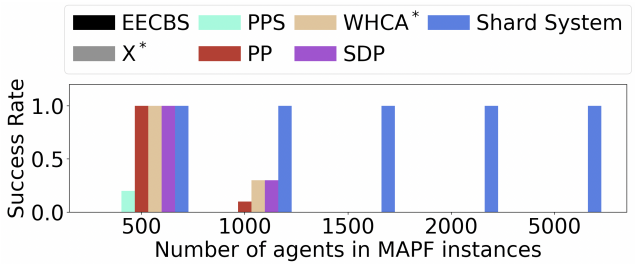

Figure 4: Comparing the shard system's scalability to that of 6 MAPF solvers on the workspace Boston_0_256.

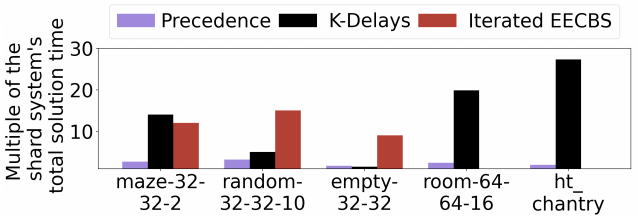

Figure 5: Comparing the shard system's ability to perform robust MAPF to that of 3 robust MAPF solvers.

challenging for search-based solvers to run in real-time and degrades the solution quality of anytime solvers.

The shard system's mean sum-of-costs was between 125% and 170% of the optimal. The shard system's global controller distributes traffic relatively evenly across its shards, reducing congestion, while its locally optimal MAPF controllers reduce the impact of bottlenecks and congestion on solution quality. The need for agents to detour through buffers, however, degrades its solution quality.

SDP and WHCA*'s mean sums-of-costs were between 150% and 1000% of the optimal. There were very few corridors in each workspace, prompting SDP to assign the majority of workspace to a single WHCA* controller. SDP and WHCA* thus performed similarly. PP's mean sum-of-costs was between 200% and 1200% of the optimal. Routing agents one by one prevents SDP, WHCA* and PP from taking global traffic patterns into account, leading to local congestion. PPS's mean sum-of-costs was between 250% and 1500% of the optimal. The large number of conflicts a load factor of 0.125 generated forced PPS's agents to take circuitous routes.

Only the shard system terminated on any large workspace. As Figure 3 (bottom) shows, its mean sum-of-costs was between 125% and 170% of the optimal. (The optimal sum-of-costs was estimated for large workspaces as the sum of the length of each agent's shortest path to its goal.)

Scalability

Next, we compare the shard system's scalability to that of the 6 MAPF solvers on the one-shot MAPF problem.

Methodology. We generated a set of 10 MAPF instances for the workspace Boston_0_256 with 500, 1,000, 1,500 and 5,000 agents. Start and goal vertices were selected as in Section . We measured each MAPF solver's *success rate* on each set, the fraction of instances it solved in 20 seconds.

Results. Figure 4 shows the experiment's results. The shard system (blue) was the only MAPF solver to have a 100% success rate on every set of instances. Its highly scalable global controller and its ability to operate its shards with perfect parallelism allow it to scale further than any other MAPF solver. EECBS (black) and X* (grey) had a 0% success rate on every set of instances. The size of a large MAPF instance's space of possible solutions makes it challenging for search-based solvers to run in real time. PPS (cyan) had a 20% and 0% success rate on the 500 and 1000+ agent MAPF instances. The large number of conflicts a large MAPF instance generates takes PPS time to resolve.

PP (red) had a 100%, 10% and 0% success rate on the 500, 1000 and 1500+ agent MAPF instances. It takes time to find a path which avoids the large number of moving obstacles prioritized planning creates for a large team. SDP (purple) and WHCA* (brown) both had a 100%, 30% and 0% success rate on the 500, 1000 and 1500+ agent MAPF instances. Boston_0_256 has very few corridors, prompting SDP to assign the majority of the workspace to a single WHCA* controller. SDP and WHCA* thus performed identically.

Robustness

Next, we compare the shard system's ability to perform robust MAPF in the presence of probabilistic delays to 3 state-of-the-art robust MAPF solvers:

1. *Precedence* (Hönig et al. 2016). A solution is generated by EECBS. If an agent is delayed, it continues to follow its path. An agent a_i may only enter a vertex v if every agent scheduled to enter v before a_i has done so already.
2. *K-Delays* (Atzmon et al. 2018). A solution where no two agents enter a vertex within $k = 4$ timesteps of each other is generated by a custom-built bounded-suboptimal CBS solver. If an agent's move actions fail $k + 1$ times, the solver is rerun.
3. *Iterated EECBS* (Li, Ruml, and Koenig 2021). EECBS is rerun every time an agent is delayed.

Methodology. Our evaluation was conducted on the sets of MAPF instances described in Section . Move actions fail 1% of the time. The performance of a robust MAPF solver depends upon both its solution quality and the speed that it can replan after a move action fails. We construct a metric which takes both factors into account as follows:

Each MAPF solver is given 5 minutes to plan an initial solution. We then simulate its solution at the rate of one timestep every 0.5 seconds. If a delay occurs, a MAPF solver may pause the simulator to replan. An agent a_i's *task time* is the time the agent takes to reach its goal. Let the time a MAPF solver needs to replan for timestep j be denoted d_j. An agent a_i's task time t_i is thus 0.5 seconds times its path length $|p_i|$ plus the total amount of time the MAPF solver takes to replan before the agent reaches its goal:

$$t_i := 0.5|p_i| + \sum_{j=1}^{|p_i|} d_j$$

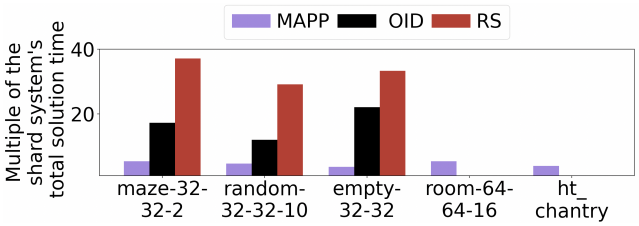

Figure 6: Comparing the shard system's ability to perform persistent MAPF to that of 3 persistent MAPF solvers.

A MAPF solver's *total solution time* T_i is the sum of its agents' task times $\sum_i t_i$. Total solution time can be thought of as a real time equivalent to sum-of-costs.

Results. Figure 5 shows each MAPF solver's total solution time on each small workspace as a multiple of the shard system's total solution time. The shard system outperformed Precedence (purple) by between 70% and 320%. Unlike the shard system, Precedence struggled to route agents past bottlenecks where a single delay could affect many agents.

The shard system outperformed K-Delays (black) by 150% to 2,700%. While K-Delays rarely had to replan, each replan took 2-10 minutes. When K-Delays didn't have to replan, the shard system's solution time was still superior because K-Delays forces agents to wait for long periods of time to access commonly used vertices.

The shard system outperformed iterated EECBS (red) by 900% to 1,500% on workspaces with less than 150 agents. Iterated EECBS failed to terminate on workspaces with more than 150 agents. Iterated EECBS reruns EECBS after every delay, which is a time-consuming process.

Persistence

Next, we compare the shard system's ability to perform persistent MAPF to 3 state-of-the-art persistent MAPF solvers: a rule-based MAPF solver, *MAPP* (Wang and Botea 2011), and two search-based solvers, *Replan Single (RS)* and *Online Independence Detection (OID)* (Švancara et al. 2019).

Methodology. We generated a MAPF instance with a load factor of $\alpha = 0.125$ for each small workspace. Each agent was assigned a start vertex and an ordered list of 4 goal vertices selected randomly from all vertices not part of a buffer. Whenever an agent reaches its current goal, it is assigned the next goal in its list. We measured each solver's total solution time as in Section .

Results. Figure 6 shows each MAPF solver's total solution time on each small workspace as a multiple of the shard system's total solution time. The shard system outperformed MAPP (purple) by 370% to 550%. The shard system's locally optimal routing achieves better quality routing in dense conditions than a rule-based approach can provide. The shard system outperformed RS (black) by 2900% to 3700% and OID (red) by 1200% to 2200%. These MAPF solver's replanning mechanisms took several seconds or more - a significant penalty to pay when a new goal is assigned.

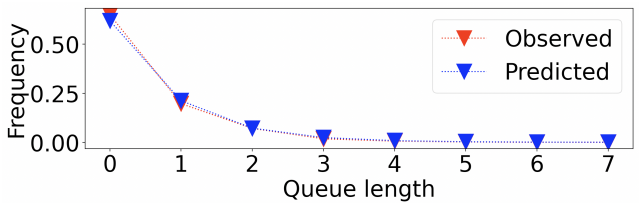

Figure 7: Comparing our model's predicted queue length distribution to the queue length distribution in the persistent MAPF experiment on the workspace maze-32-32-2.

Buffer Queue Modeling

Finally, we assess how well a $M/M/1/n_{buf}/FCFS$ queue describes the behavior of the shard system's buffers.

Methodology. During the persistent MAPF experiment, we selected a random buffer in each workspace and measured its queue length at each timestep. We compare our model's predicted queue length distribution to the observed queue length distribution using the reduced χ^2 statistic. The reduced χ^2 statistic is computed with n_{buf} degrees of freedom: one degree of freedom for each of the $n_{buf}+1$ possible queue lengths a buffer of length n_{buf} can contain, minus one because the density factor ν is effectively a fitted parameter.

Results. When the value of α in our model is adjusted by a density factor of $\nu = 1.33$ (to account for a greater average density of agents near the mouth of a queue), we obtain reduced χ^2 values ranging from 1.32 to 2.11, suggesting that our model is far from a perfect fit but still a good first approximation. Figure 7 plots the frequency with which the buffer in maze-32-32-2 we examined had a particular queue length in red and the number of times our queuing model predicts a particular queue length in blue. For each queue length, the predicted frequency is within 10% of the observed frequency.

Conclusion

To conclude, in this paper we presented the shard system, a novel MAPF solver which can perform robust, persistent MAPF for thousands of agents in real time. The shard system can also solve instances of the one-shot MAPF problem for teams of thousands of agents on a diversity of workspaces in seconds. Its solutions' sum-of-costs, moreover, were within 125% to 170% of the optimal. A shard system can replan for thousands of agents in real time because its workspace partitioning algorithm produces shards which contain a small number of agents and its buffers allow these shards to replan with perfect parallelism.

Acknowledgements

This research at the University of Southern California was supported by the National Science Foundation (NSF) under grant numbers 1409987, 1724392, 1837779, and 1935712 as well as a gift from Amazon. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the sponsoring organizations, agencies, or the U.S. government.

References

A., K.; and R., H. 2004. Balanced Graph Partitioning. *The Annual ACM Symposium on Parallelism in Algorithms and Architectures*, 120–124.

Atzmon, D.; Stern, R.; Felner, A.; Wagner, G.; Barták, R.; and Zhou, N.-F. 2018. Robust Multi-Agent Path Finding. *The Symposium on Combinatorial Search*, 2–9.

Barer, M.; Sharon, G.; Stern, R.; and Felner, A. 2014. Suboptimal Variants of the Conflict-Based Search Algorithm for the Multi-Agent Pathfinding Problem. *The Symposium on Combinatorial Search*, 19–27.

Boyarski, E.; Felner, A.; Sharon, G.; and Stern, R. 2015. Don't Split, Try To Work It Out: Bypassing Conflicts in Multi-Agent Pathfinding. *The International Conference on Automated Planning and Scheduling*, 47–51.

Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; and Stein, C. 2009. *Introduction to Algorithms*. The MIT Press, 3rd edition.

De Wilde, B.; Ter Mors, A. W.; and Witteveen, C. 2014. Push and Rotate: A Complete Multi-Agent Pathfinding Algorithm. *Journal of Artificial Intelligence Research*, 51(1): 443–492.

Google. 2021. OR-Tools. https://developers.google.com/optimization. Accessed: 2022-04-28.

Gross, D.; Shortle, J. F.; Thompson, J. M.; and Harris, C. M. 2008. *Fundamentals of Queueing Theory*. Wiley-Interscience, 4th edition.

Hönig, W.; Kumar, T. K. S.; Cohen, L.; Ma, H.; Xu, H.; Ayanian, N.; and Koenig, S. 2016. Multi-Agent Path Finding with Kinematic Constraints. *The International Conference on Automated Planning and Scheduling*, 477–485.

Hönig, W.; Kiesel, S.; Tinka, A.; Durham, J. W.; and Ayanian, N. 2019. Persistent and Robust Execution of MAPF Schedules in Warehouses. *IEEE Robotics and Automation Letters*, 4(2): 1125–1131.

Karypis, G.; Aggarwal, R.; Kumar, V.; and Shekhar, S. 1998. Multilevel Hypergraph Partitioning: Application in VLSI Domain. *Proceedings of the 34th Annual Design Automation Conference*, 526–529.

Kendall, D. 1953. Stochastic Processes Occurring in the Theory of Queues and their Analysis by the Method of the Imbedded Markov Chain. *Annals of Mathematical Statistics*, 24: 338–354.

Li, J.; Gange, G.; Harabor, D.; Stuckey, P. J.; Ma, H.; and Koenig, S. 2020. New Techniques for Pairwise Symmetry Breaking in Multi-Agent Path Finding. *The International Conference on Automated Planning and Scheduling*, 193–201.

Li, J.; Ruml, W.; and Koenig, S. 2021. EECBS: A Bounded-Suboptimal Search for Multi-Agent Path Finding. *The AAAI Conference on Artificial Intelligence*, 35: 12353–12362.

Lozano-Pérez, P.; and Erdmann, M. 1987. A novel approach to path planning for multiple robots in bi-connected graphs. *Algorithmica*, 2: 477–521.

Ma, H.; Li, J.; Kumar, T. K. S.; and Koenig, S. 2017. Lifelong Multi-Agent Path Finding for Online Pickup and Delivery Tasks. *The International Conference on Autonomous Agents and Multiagent Systems*, 837–845.

Pianpak, P.; Son, T. C.; Toups, Z. O.; and Yeoh, W. 2019. A Distributed Solver for Multi-Agent Path Finding Problems. *The International Conference on Distributed Artificial Intellgence*, 1–7.

Raghavan, P.; and Tompson, C. D. 1987. Randomized rounding: A technique for provably good algorithms and algorithmic proofs. *Combinatorica*, 7(4): 365–374.

Sajid, Q.; Luna, R.; and Bekris, K. E. 2012. Multi-Agent Pathfinding with Simultaneous Execution of Single-Agent Primitives. *The Symposium on Combinatorial Search*, 88–96.

Schranz, M.; Umlauft, M.; Sende, M.; and Elmenreich, W. 2020. Swarm Robotic Behaviors and Current Applications. *Frontiers in Robotics and AI*, 7(36): 1–12.

Sharon, G.; Stern, R.; Felner, A.; and Sturtevant, N. R. 2012. Conflict-Based Search for Optimal Multi-Agent Path Finding. *Artificial Intelligence*, 219: 40–66.

Silver, D. 2005. Cooperative Pathfinding. *The AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 117–122.

Simon, M. 2019. Inside the Amazon Warehouse Where Humans and Machines Become One. https://www.wired.com/story/amazon-warehouse-robots. Accessed: 2022-04-28.

Stern, R.; Sturtevant, N. R.; Felner, A.; Koenig, S.; Ma, H.; Walker, T. T.; Li, J.; Atzmon, D.; Cohen, L.; Kumar, T. K. S.; Barták, R.; and Boyarski, E. 2019. Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks. *The International Symposium on Combinatorial Search*, 151–159.

Vedder, K.; and Biswas, J. 2019. X*: Anytime Multiagent Path Planning With Bounded Search. *The Conference on Autonomous Agents and Multi-Agent Systems*, 2247–2249.

Švancara, J.; Vlk, M.; Stern, R.; Atzmon, D.; and Barták, R. 2019. Online Multi-Agent Pathfinding. *The AAAI Conference on Artificial Intelligence*, 33(1): 7732–7739.

Wang, C. K.; and Botea, A. 2011. MAPP : a Scalable Multi-Agent Path Planning Algorithm with Tractability and Completeness Guarantees. *Journal of Artificial Intelligence Research*, 42: 55–90.

Wilt, C. M.; and Botea, A. 2014. Spatially Distributed Multiagent Path Planning. *The International Conference on Automated Planning and Scheduling*, 332–340.

Yu, J.; and LaValle, S. M. 2016a. Optimal multirobot path planning on graphs: Complete algorithms and effective heuristics. *IEEE Transactions on Robotics*, 32(5): 1163–1177.

Yu, J.; and LaValle, S. M. 2016b. Optimal multirobot path planning on graphs: Complete algorithms and effective heuristics. *IEEE Transactions on Robotics*, 32: 1163–1177.

Zhang, H.; Yao, M.; Liu, Z.; Li, J.; Terr, L.; Chan, S.-H.; Kumar, S.; and Koenig, S. 2021. A Hierarchical Approach to Multi-Agent Path Finding. *The ICAPS-21 Workshop on Hierarchical Planning*.