# Fault-Tolerant Offline Multi-Agent Path Planning

**Keisuke Okumura**[1], **Sébastien Tixeuil**[2]

[1] Tokyo Institute of Technology, Japan
[2] Sorbonne University, CNRS, LIP6, Institut Universitaire de France, France
okumura.k@coord.c.titech.ac.jp, Sebastien.Tixeuil@lip6.fr

## Abstract

We study a novel graph path planning problem for multiple agents that may crash at runtime, and block part of the workspace. In our setting, agents can detect neighboring crashed agents, and change followed paths at runtime. The objective is then to prepare a set of paths and switching rules for each agent, ensuring that all correct agents reach their destinations without collisions or deadlocks, despite unforeseen crashes of other agents. Such planning is attractive to build reliable multi-robot systems. We present problem formalization, theoretical analysis such as computational complexities, and how to solve this offline planning problem.

## 1 Introduction

Building robust and resilient multi-robot systems is an emerging and important topic (Prorok et al. 2021), since those systems are expected to be infrastructures of logistics or product lines, as seen in fleet operations in automated warehouses (Wurman, D'Andrea, and Mountz 2008). One fundamental problem in multi-robot systems is multi-agent path planning (MAPP), which assigns a collision/deadlock-free path to each agent. Therefore, designing robust and resilient approaches to MAPP is a critical component to realize reliable multi-robot systems.

*Robot faults* are not rare in practice due to sensor/motor errors, battery consumption, or other unexpected events. Here, "faults" are beyond delays of robot motions as studied in the MAPP literature (Atzmon et al. 2020; Shahar et al. 2021; Okumura et al. 2022). Rather, we consider them as critical events, e.g., agents forever stop their motions due to crashes, or, misbehave from the planning. Then, fault-tolerant properties are essential for building reliable multi-robot systems, especially in lifelong scenarios involving a large number of robots. Nevertheless, the cutting-edge MAPP studies largely overlook this aspect and assume that agents perfectly follow the offline planning that is prepared without any fault.

To this end, as the first step of fault-tolerant MAPP, *we study an MAPP problem where agents may unexpectedly crash at runtime*. The crashed agents then forever block part of the workspace. Correct agents (i.e., non-crashed ones) can detect crashes through *local observations* and then switch their executing path on the fly, based on this observation. Our objective is to find a set of paths and their switching rules for each agent, such that correct agents can reach their destinations regardless of crash patterns. We refer to the corresponding offline planning problem as *MAPP with Crash Faults (MAPPCF)*.

Throughout the paper, we rely on local observations (rather than global observations) that permit to immediately detect a crash if it occurs on a neighboring location. Therefore, significantly different from conventional MAPP studies, the challenge is to design a safe planning methodology that avoids collisions and deadlocks, under the assumption that agents behave following their plan, and their own observed information about other agents' crashes.

The contributions of this paper are twofold:

**We formalize and analyze MAPPCF.** Our formalization includes the conventional synchronous execution model of multi-agent pathfinding (MAPF) (Stern et al. 2019) where all agents take actions simultaneously, as well as the recently proposed asynchronous execution model called OTIMAPP (Okumura et al. 2022). In our model, agents can switch executing paths on the fly according to local observation results. The observation is done using a *failure detector*, a black box function that tells an agent whether an adjacent location is occupied by a crashed agent, occupied by a correct agent, or vacant. We consider anonymous and named failure detectors; the former cannot identify a crashed agent (only a crashed location). After characterizing relationships between execution model and failure detector variants, we analyze the computational complexities of MAPPCF. Our main results are that finding a solution is NP-hard, and verifying a solution is co-NP-complete.

**We propose a methodology to solve MAPPCF.** The proposed method, *decoupled crash faults resolution framework (DCRF)*, resolves the effects of crashes one by one by preparing backup paths. We evaluate DCRF with named failure detectors in grid environments and observe that DCRF can address more problem instances compared to computing a set of vertex disjoint paths, i.e., a trivially fault-tolerant approach since correct agents can reach their destinations regardless of crash patterns. We further observe that the difficulty of finding solutions stems both from the problem in-

stance size (e.g., the number of agents), and from the number of crashes to be tolerated.

The paper organization is as follows. Section 2 formalizes MAPPCF. Section 3 and 4 present preliminary and computational complexity analysis, respectively. Section 5 describes DCRF. Section 6 presents empirical results obtained with DCRF. Section 7 reviews related work. Section 8 concludes the paper with discussion of future directions. The supplementary material is available on https://kei18.github.io/mappcf. This paper uses "MAPP" as a generalized term for path planning for multiple agents, not limited to the formalization of classical MAPF.

## 2 Problem Definition

**MAPPCF Instance** An *MAPPCF instance* is given by a graph $G = (V, E)$, a set of agents $A = \{1, 2, \dots, n\}$, the maximum number of crashes $f \in \mathbb{N}_{\geq 0}$, a tuple of starts $(s_1, s_2, \dots, s_n)$, and goals $(g_1, g_2, \dots, g_n)$, where $s_i, g_i \in V$ and for all $i \neq j$, $s_i \neq s_j$ and $g_i \neq g_j$. An MAPPCF instance on digraphs is similar to the undirected case.

**Plan** A *plan* for one agent comprises a list of paths each defined on $G$ and *transition rules*. At runtime, the agent moves along one path, called *executing path*, in the plan, while always occupying one vertex. Meanwhile, the agent switches its executing path following the transition rules. A plan contains one special path called *primary path* which is initially executed. A transition rule is defined with a *failure detector* and *progress index*, explained below.

**Crash and Failure Detector** During plan execution, agents are potentially crashed. *Crashed* agents eternally remain in their occupying vertices. We refer *correct* agents to those who are not crashed. Correct agents cannot pass where crashed agents are located. However, a correct agent can use a *failure detector* at runtime to change its executing path. Doing so enables the correct agent to reach its goal under crash faults. A failure detector tells a correct agent about the existence of an agent on an adjacent vertex, and if so, whether it has crashed or not. We consider two types of detectors. A detector is called *named (NFD)* when it can identify who is crashed, otherwise *anonymous (AFD)*. Formal definitions are as follows. Assume that an agent $i$ is at $v \in V$. Let denote $\mathcal{N}_v$ a set of adjacent vertex of $v$. Then, $\mathsf{AFD} : \mathcal{N}_v \mapsto \{\circ, \times, \bot\}$ and $\mathsf{NFD} : \mathcal{N}_v \mapsto \{\circ, \bot\} \cup A$. Here, $\circ$ and $\times$ respectively correspond to a correct or crashed agent, otherwise $\bot$ is returned (no agent is there). NFD returns an agent instead of $\times$.

**Progress Index** We use a *progress index* $clock_i \in \mathbb{N}_{>0}$; the agent $i$ is at $\langle clock_i \rangle$–th vertex of its executing path. For each transition of executing paths, the progress index is initialized with one. It increases up to the length of the executing path.

**Transition Rule** The rule specifies the next executing path given the current executing path, progress index, and results of the failure detector. We then describe two execution models that differ in how to increment progress indexes.

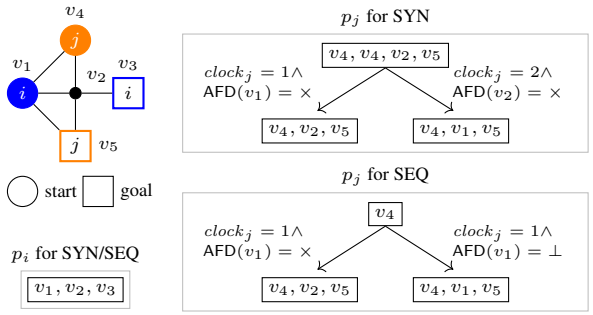



Figure 1: Solution example with AFD.

**Synchronous Execution Model (SYN)** In this model, all agents take actions simultaneously. More precisely, all the correct agents perform the following at the same time: *(i)* may crash, *(ii)* change executing paths if necessary, *(iii)* move to their next vertices, and *(iv)* increment their progress indexes. Two types of collisions must be prohibited by plans: *a)* vertex collisions, i.e., two agents are on the same vertex simultaneously, and *b)* swap collisions, i.e., two agents swap their hosting vertices simultaneously. Note that an agent can remain at its hosting vertex if its executing path contains the same vertex consecutively.

**Sequential Execution Model (SEQ)** In this model, agents take actions sequentially but we cannot control how agents are scheduled at runtime. More precisely, given an infinite sequence of agents $\mathcal{E}$ called *execution schedule*, the agents are *activated* in turn according to $\mathcal{E}$. An activated agent performs the following: *(i)* change executing paths if necessary, *(ii)* move to its next vertex specified by the progress index if the vertex is unoccupied by other agents, and *(iii)* increment its progress index if moved. The agent remains on its hosting vertex when the next vertex is occupied. Agents may crash at any time, except for the duration of the procedures for activation. $\mathcal{E}$ is unknown when offline planning, but we assume that every agent appears infinitely-many times in $\mathcal{E}$.

**Solution** Given an MAPPCF instance, a *solution for SYN* is a set of plans $\{p_1, p_2, \dots, p_n\}$ respectively for each agent, such that:

1. The primary path of $p_i$ begins with a start $s_i$.
2. For each path that is not primary, the path begins with a vertex where the agent changes its executing path.
3. The agent $i$ is ensured to reach its goal $g_i$ provided that $i$ follows $p_i$, regardless of other agents' crashes, when the total number of crashes is up to $f$.

A *solution for SEQ* is similar to SYN but the third condition should be satisfied for any execution schedule. Figure 1 presents solution examples for both models. Without crash assumptions (i.e., if $f = 0$), solutions for SYN are equivalent to those of classical MAPF (Stern et al. 2019), and solutions for SEQ are equivalent to those of OTIMAPP (Okumura et al. 2022). We assume $f > 0$ in the reminder.

**Remarks** An index of sequences starts at one. In SYN, a solution must prevent collisions. In SEQ, agents are as-

sumed to follow planned paths, while avoiding collisions locally (e.g., by adjusting their velocity). Rather, a solution must prevent deadlocks, wherein several agents block their progress from each other. The detailed analyses for SEQ appear in (Okumura et al. 2022). Implementations of failure detectors depend on applications, e.g., using heartbeats as commonly used in distributed network systems (Felber et al. 1999) or multi-robot platforms such that environments can detect robot faults (Kameyama et al. 2021). Herein, failure detectors are assumed to be black-box functions.

## 3 Preliminary Analysis

We first present two fundamental analyses to grasp the characteristics of MAPPCF: the model power and the necessary condition for instance to include a solution.

### 3.1 Model Power

A model $X$ is *weakly stronger* than another model $Y$ when all solvable instances in $Y$ are also solvable in $X$. $X$ is *strictly stronger* than $Y$ when it is weakly stronger than $Y$ and there exists an instance that is solvable in $X$ but unsolvable in $Y$. Two models are *equivalent* when both are respectively weakly stronger than another.

A *model* of MAPPCF is specified by two components: *(i)* whether the failure detector is anonymous (AFD) or named (NFD), and *(ii)* whether the execution model is synchronous (SYN) or sequential (SEQ). Characterizing model power, i.e., which model is stronger than another, is important because it can be instrumental when implementing the algorithm. For instance, AFD is intuitively easier to implement than NFD. So, if those two models have equivalent power, then we may not need to realize NFD.

The main results are summarized in Fig. 2. Several relationships are still open questions, e.g., whether NFD is strictly stronger than AFD in SYN. In what follows, we present three theorems for the model power analysis.


Figure 2: Relationship of models. '$X \rightarrow Y$' denotes that model $Y$ is strictly stronger than model $X$. A dashed arrow means weakly stronger relationship.

**Theorem 1.** *When using the same execution model, NFD is weakly stronger than AFD.*

*Proof.* NFD can emulate AFD by dropping "who." □

**Theorem 2.** *When using the same failure detector types, SYN is strictly stronger than SEQ.*

*Proof.* Figure 3 shows an instance that is solvable for SYN but unsolvable for SEQ. In SYN, the agent $j$ can wait until $i$ passes the middle two vertices, and according to crash patterns, $j$ can change its path towards its goal. However, in SEQ, there are execution schedules that $j$ enters either of the middle two vertices prior to $i$ because $j$ cannot distinguish whether $i$ is on its start or goal. If $j$ is crashed there and $i$ still remains at its start, $i$ cannot reach its goal.

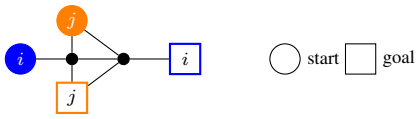



Figure 3: Solvable instance in SYN but unsolvable in SEQ.

Next, we prove that every solvable instance in SEQ is solvable in SYN. Consider constructing a new solution $Z_{\text{syn}}$ in SYN given a solution $Z_{\text{seq}}$ in SEQ. This is achieved by, considering one execution schedule (e.g., (1, 2, …, n, 1, 2, …, n, …)) and allowing $Z_{\text{syn}}$ to move agents in their turn. For instance, at timestep one, only agent-1 is allowed to move, and at timestep two, only agent-2 is allowed to move, and so forth. With appropriate modifications of paths specified $Z_{\text{seq}}$, since $Z_{\text{seq}}$ solves the original instance, $Z_{\text{syn}}$ also solves the instance in SYN. □

**Theorem 3.** *In SEQ, NFD is strictly stronger than AFD.*

*Proof sketch.* We show an instance that is only solvable with NFD in SEQ in the Appendix. □

### 3.2 Necessary Condition

**Theorem 4.** *Regardless of execution models and failure detector types, two conditions are necessary for instances to contain solutions.*

- *No use of other goals: For each agent $i$, there exists a path in $G$ from $s_i$ to $g_i$ that does not include any $g_j$, for all $j \in A, j \neq i$.*
- *Limitation of other starts: For each agent $i$, for each $B \subset A$ where $i \notin B$ and $|B| = f$, there exists a path in $G$ from $s_i$ to $g_i$ that does not include $s_j$, for all $j \in B$.*

*Proof. No use of other goals*: Suppose that there exists an agent $i$ that needs to pass through one of the goals $g_j$. If $i$ crashes at $g_j$, then $j$ cannot reach $g_j$. *Limitation of other starts*: Suppose that there exists an agent $i$ that needs to pass through one of the starts of $B$. If all agents in $B$ are crashed at their starts, then $i$ cannot reach $g_i$. □

When $f = |A| - 1$, the conditions in Thrm. 4 are equivalent to a *well-formed instance* (Čáp et al. 2015) for MAPF; an instance such that every agent has at least one path that uses no others' start and goal vertices. Our condition slightly differs in the limitation of starts because agents can change their behavior at runtime, according to failure detectors.

## 4 Computational Complexity

This section discusses the complexity of MAPPCF, namely, studying two questions: the difficulty of finding solutions and that of verifying solutions. The primary result is that both problems are computationally intractable; the former is NP-hard and the latter is co-NP-complete. Both proofs are based on reductions of the 3-SAT problem, determining the

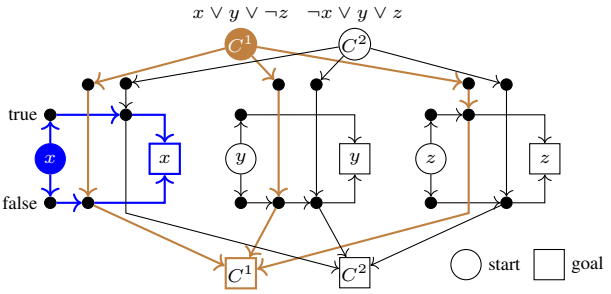
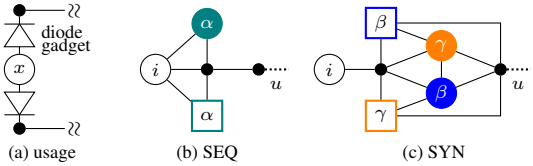



Figure 4: MAPPCF instance on a directed graph in SEQ reduced from the SAT instance $(x \vee y \vee \neg z) \wedge (\neg x \vee y \vee z)$.




(a) usage (b) SEQ (c) SYN

Figure 5: Diode gadgets.

satisfiability of a formula in conjunctive normal form with three literals in each clause. The proof of verification appears in the Appendix.

**Theorem 5.** *MAPPCF is NP-hard regardless of models.*

*Proof.* We first prove that MAPPCF on digraphs in SEQ is NP-hard. The proof is done by reduction of the SAT problem and works regardless of failure detector types. Throughout the proof, we use the following example: $(x \vee y \vee \neg z) \wedge (\neg x \vee y \vee z)$. The reduction is depicted in Fig. 4.

*A. Construction of an MAPPCF Instance:* We first introduce a *variable agent* for each variable $x_i$. In Fig. 4, we highlight the corresponding agent of the variable $x$ as blue-colored. The reduced instance has two paths for each variable agent: *upper* or *lower* paths. Both paths include at least one vertex (just above/below of the start in Fig. 4) and additional vertices depending on clauses of the formula.

We next introduce a *clause* agent for each clause $C^j$ of the formula. Each clause agent has multiple paths, corresponding to each literal in the clause. Those paths contain two vertices excluding the start and the goal: one vertex unique to each literal, and another vertex shared with the corresponding variable agent. The shared vertex is located on the lower (or upper) path of the variable agent when the literal is positive (resp. negative). In Fig. 4, we highlight the corresponding agent of the clause $C^1 = x \vee y \vee \neg z$ as brown-colored. The translation from the formula into an MAPPCF instance is clearly done in polynomial time.

*B. MAPPCF has a solution if the formula is satisfiable:* Given a satisfiable assignment, a solution of MAPPCF is built as follows. When a variable is assigned true (or false), let the corresponding agent takes the upper (resp. lower) path. Each clause agent then has at least one path (vertex) disjoint with any variable agent; otherwise, the clause is unsatisfied. Let the clause agent take this path; these paths constitute a solution because all paths are disjoint.

*C. The formula is satisfiable if MAPPCF has a solution:* In every solution, a plan for each agent inherently consists of a single path, due to the instance construction. These paths should be (vertex) disjoint; otherwise, the crash of one agent blocks another from reaching its goal. Then, build an assignment as follows. Assign a variable true (or false) when the corresponding variable agent uses the upper (resp. lower) path. This assignment is satisfiable because it ensures at least one literal is satisfied in all clauses.

*D. Extending the reduction to undirected graphs:* The aforementioned proof is extended to the undirected case by introducing a *diode* gadget to the starts of every agent, as partially shown in Fig. 5a. This gadget prevents back to the start once the agent passes through the gadget (i.e., reaching vertex $u$ in Fig. 5b–c). Therefore, the same proof procedure (i.e., finding disjoint paths) is applied to other models. The gadget for SEQ and SYN are shown in Fig. 5b–c, respectively. The proofs of their properties are delivered in the Appendix. □

**Theorem 6.** *Verifying a solution of MAPPCF is co-NP-complete regardless of models.*

Although MAPPCF is intractable in general, it is notable that the difficulties might be relaxed in certain classes, e.g., instances on planner graphs or when $f = 1$.

## 5 Solving MAPPCF

We next discuss how to solve MAPPCF. The main challenge is how to manage crash awareness differences among agents. For instance, agent $i$ may observe a crash of agent $j$ while at a neighboring position, and change its path accordingly. However, another correct agent $k$, located further away from $j$, might not be aware that $j$ has crashed. To preserve safety, a plan of $k$ requires avoiding collisions and deadlocks with both before-after paths of $i$ (that is, regardless of the crash of $j$). Any planning algorithm solving MAPPCF thus has to cope with different awareness of crash patterns.

The proposed method, *decoupled crash faults resolution framework (DCRF)*, returns an MAPPCF solution. Herein, we consider only NFD, but DCRF is also applicable to AFD.

### 5.1 Framework Description

Algorithm 1 presents DCRF. Figure 6 illustrates a running example in SYN. We first describe DCRF in SYN using this example, followed by detailed parts of the implementation.

**Finding Initial Plans** DCRF first obtains an initial plan for each agent (i.e., a path) [Line 1]. This process is equivalent to solving MAPF with existing solvers. In the example, the initial plans for agents $i$, $j$, and $k$ are respectively depicted in Fig. 6d, 6b, and 6c.

**Identifying Unresolved Events** DCRF next identifies *unresolved events*. An event is a pair of *crash* (i.e., who crashes where and when), and *effect* (i.e., whose path is affected where and when). Finding unresolved events is done by finding shared vertices in a set of paths. In the example (Fig. 6d), the initial plans contain two unresolved events:

Algorithm 1: DCRF

**input**: instance $\mathcal{I}$; **output**: solution $\mathcal{P}$ or FAILURE

1: $\mathcal{P} \leftarrow$ get_initial_plans($\mathcal{I}$)
2: $\mathcal{U} \leftarrow$ get_initial_unresolve_events($\mathcal{I}, \mathcal{P}$)
3: **while** $\mathcal{U} \neq \emptyset$ **do**
4: $\quad e \leftarrow \mathcal{U}$.pop() ▷ *event; pair of crash & effect*
5: $\quad \pi \leftarrow$ find_backup_path($\mathcal{I}, \mathcal{P}, e$)
6: $\quad$ **if** $\pi$ not found **then return** FAILURE
7: $\quad \mathcal{U}$.push(get_new_unresolved_events($\mathcal{I}, \mathcal{P}, \pi$))
8: $\quad$ update $\mathcal{P}$ with $\pi$
9: **return** $\mathcal{P}$

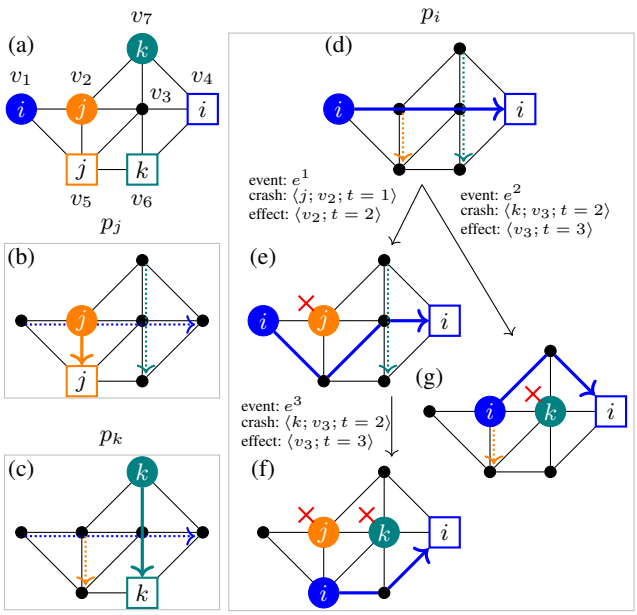



Figure 6: Running example of DCRF in SYN. A red cross corresponds to an observed crashed agent. Dotted lines are paths with which the planning should avoid collisions.

- $e^1$: $p_i$ cannot use $\langle v_2; t = 2\rangle$ if $j$ is crashed at $\langle v_2; t = 1\rangle$
- $e^2$: $p_i$ cannot use $\langle v_3; t = 3\rangle$ if $k$ is crashed at $\langle v_3; t = 2\rangle$

These events should be resolved by preparing *backup paths*. DCRF resolves events in a decoupled manner as follows.

**Resolving Events** Unresolved events are stored in a priority queue $\mathcal{U}$ and handled one by one [Lines 3–8]. For each event, DCRF tries to find a backup path [Line 5]. This is a single-agent pathfinding problem, whose goal location is the same as the initial path, while the start location is one-step before where the crashed agent is. The pathfinding is constrained to avoid collisions with observed crashed agents and other already constructed plans. If failed to find such a path, DCRF reports FAILURE. Otherwise, DCRF identifies new unresolved events with the backup path, and updates the plan. The event is now resolved. When all events are resolved, the framework returns a solution [Line 9]. In the example, when the event $e^1$ is popped from $\mathcal{U}$, DCRF computes the backup path shown in Fig. 6e. Observe that this backup path must not use $v_6$ to avoid collisions with $p_k$. DCRF then identifies and registers a new unresolved event $e^3$, and updates $i$'s plan. DCRF continues applying the same procedure for the events $e^2$ (Fig. 6g) and $e^3$ (Fig. 6f).

**Remark** DCRF is incomplete, i.e., it does not guarantee to return a solution even though an instance is solvable. Such a failure example is available in the Appendix.

### 5.2 Implementation Details

**Discarding Unnecessary Events** When one crash affects a given path several times, only the first effect should be resolved. This happens when the path is not simple. For this purpose, the queue $\mathcal{U}$ stores events in ascending order of their occurring time. DCRF then discards events when encountering already resolved crashes.

**Inconsistent Crash Patterns** When preparing a backup path for agent $i$, pathfinding does not necessarily need to avoid collisions with all others' paths. For instance, Fig. 6e assumes $j$ has crashed. Therefore, in descendant backup paths for $i$, $i$ can neglect $j$'s plan. Similarly, when two different paths assume crashes at distinct locations of the same agent, or when two paths assume more than $f$ crashes in total, those two paths cannot be executed during the same execution. Consequently, DCRF does not need to identify unresolved events between these paths.

**Refinement of Initial Paths** Reducing the number of events is crucial for constructing solutions, as a higher number of events yields a higher number of paths for each agent. Consequently, when preparing a backup path, the pathfinding process tries to avoid collisions with those many paths, and may fail to find a suitable path. To circumvent this problem, when preparing initial plans, we introduce a refinement phase that minimizes the use of shared vertices between agents. This is done by adapting the technique to improve MAPF solutions (Okumura, Tamura, and Défago 2021).

**Implementation for SEQ** DCRF is applicable to SEQ by simply replacing 'collision' and 'MAPF' by 'deadlocks' and 'OTIMAPP,' respectively.

**Implementation for AFD** For AFD, the same workflow is available by assuming that observed crashes are anonymous.

## 6 Evaluation

This section evaluates DCRF in both SYN and SEQ with NFD. We present a variety of aspects including merits to consider MAPPCF and bottlenecks of the planning.

**Baseline** We compared DCRF with a procedure to obtain pairwise vertex-disjoint paths. The rationale is that disjoint paths are trivially fault-tolerant; regardless of crash patterns, correct agents can always reach their destinations. On the other hand, with more agents, it is expected that finding such paths becomes impossible. The disjoint paths were obtained by an adapted version of conflict-based search (Sharon et al. 2015), a celebrated MAPF algorithm. The adapted one is complete, i.e., eventually returning disjoint paths if they exist, otherwise, reporting non-existence.

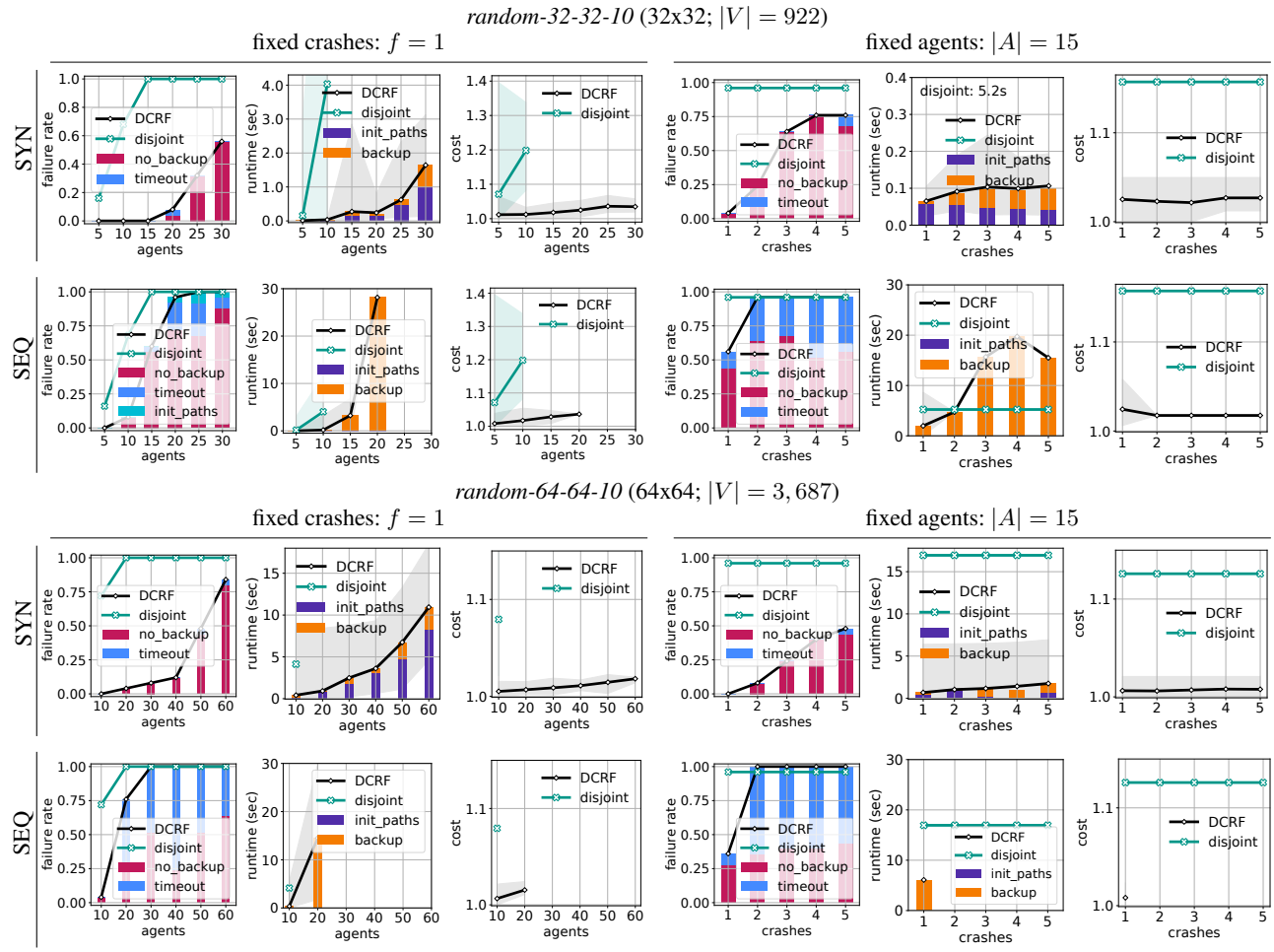



Figure 7: Summary of results. Three types of figures are included. *failure rate*: We also show failure reasons of DCRF by stacking graphs. 'no_backup' means that DCRF failed to prepare a backup path. 'timeout' means that DCRF reaches the time limit. 'init_paths' means that DCRF fails to prepare initial paths. *runtime*: The average runtime of successful instances is presented, accompanied by minimum and maximum values shown by transparent regions. We also show runtime profiling, which is categorized into preparing initial paths ('init_paths') and computing backup paths ('backup'). *cost*: This rates solution quality when crashes do not happen. See the result description for details. The average, minimum, and maximum values of the successful instances are shown. Note that finding disjoint paths are irrelevant from $f$.

**Experimental Design** MAPPCF has two critical factors: the number of agents $|A|$ and crashes $f$. To investigate those effects on the planning, we prepared two scenarios: *(i)* fixing $f$ while changing $|A|$, or *(ii)* fixing $|A|$ while changing $f$. Each scenario was tested on two four-connected grids (size: 32x32 and 64x64) obtained from (Stern et al. 2019). These grids contain randomly placed obstacles (10%). For each scenario, grid, and $|A|$, we prepared 25 well-formed instances (Čáp et al. 2015), considering the necessary condition for solutions to exist (see Sec. 3.2); however, unsolvable instances might still be included because it is not sufficient. The identical instances were used in both SYN and SEQ.

**Planning Failure** We regard that a method succeeds in solving an instance when it returns a solution before the timeout of 30 s; otherwise, the attempt is a failure.

**Implementation of DCRF** The initial paths were obtained by prioritized planning (Čáp et al. 2015; Okumura et al. 2022), respectively for SYC and SEQ. Single-agent pathfinding was implemented by A$^*$, adding a heuristic that penalizes the use of common vertices with other agents' paths. We informally observed that this improves the success rate due to a smaller number of events. We applied the refinement over initial paths (Sec. 5.2) for SYN but not for SEQ because the effect was subtle (see the Appendix).

**Evaluation Environment** The code was written in Julia and is available in the supplementary material. The experiments were run on a desktop PC with Intel Core i9-7960X 2.8 GHz CPU and 64 GB RAM. We executed 32 different instances in parallel using multi-threading.

**Results** Figure 7 shows the results. The main findings are:

► *Regardless of models, finding solutions become difficult to compute as the number of agents $|A|$ or crashes $f$ increase.* With larger $|A|$ or $f$, DCRF needs to manage a huge number of crash patterns. Consequently, DCRF often fails to find backup paths, or reaches timeout.

► *MAPPCF can address more crash situations compared to just finding disjoint paths.* Note however that the gaps in the success rate between DCRF and finding disjoint paths becomes smaller in SEQ. This is partially due to finding deadlock-free paths as they are difficult to compute in SEQ.

► *Without crashes, DCRF provides solutions with smaller costs, compared to disjoint paths.* In Fig. 7, we also present the cost of paths that agents are to follow if there is no crash. For SYN, a cost is total traveling time (aka. sum-of-costs). For SEQ, a cost is sum of path distance. Both scores were normalized by sum of distances between start-goal pairs; hence the minimum is one. Note that the cost is identical with different $f$ if $|A|$ and start-goal locations are the same. The result indicates that DCRF provides better planning that suppresses redundant agents' motions when the entire system operates correctly.

The success rate of the planning in larger grids with fixed $f = 1$ is available in the Appendix. Since this is the first study of MAPPCF where agents may observe different information at runtime, we acknowledge the room for algorithmic improvements. Therefore, we consider that further improvements of DCRF or developing new MAPPCF algorithms are promising directions. Specifically, as seen in Fig. 7, a large amount of failure reasons is failing to prepare backup paths. DCRF takes a decoupled approach that sequentially plans a path for each agent. Instead, developing coupled approaches may decrease the failure rate.

## 7 Related Work

**Path Planning for Multiple Agents** Navigation for a team of agents are typically categorized into *reactive* or *deliberative* approaches. Reactive approaches (Van Den Berg et al. 2011; Senbaslar, Hönig, and Ayanian 2018) make agents continuously react to situations at runtime to avoid collisions, while heading to their own destinations. This class can deal with unexpected events such as crash failures. However, provably deadlock-free systems are difficult to realize due to the shortsightedness of time evolution. Deliberative approaches use a longer planning horizon to plan collision/deadlock-free trajectories, typically formulated as the *multi-agent pathfinding (MAPF)* problem (Stern et al. 2019). Recent studies (Atzmon et al. 2020; Shahar et al. 2021; Okumura et al. 2022) focus on robust MAPF for timing uncertainties, i.e., where agents might be delayed at runtime. On the other hand, those studies assume that agents never crash and eventually take action; significantly different from ours. MAPPCF is on the deliberative side but also has reactive aspects because agents change their behaviors at runtime according to failure detectors.

DCRF can be regarded as a two-level search, akin to popular MAPF algorithms (Sharon et al. 2013, 2015; Surynek 2019; Lam et al. 2022). Those algorithms manage collisions at a high level, and perform single-agent pathfinding at a low level. Instead of collision management, DCRF manages unresolved crash faults at the high level.

Multi-agent path planning with *local observations* is not new in the literature (Wiktor et al. 2014; Zhang, Kim, and Fainekos 2016). Typically, previous studies aim at avoiding collisions or deadlocks by applying ad-hoc rules of agents' behavior, according to observation results at runtime, without assumptions of crash fault. We note that learning-based MAPF approaches like (Sartoretti et al. 2019) also assume local observations of each agent (e.g., field-of-view).

**Resilient Multi-Robot Systems** Studies on multi-robot systems sometimes assume robot crashes at runtime, e.g., for target tracking (Zhou et al. 2018), orienteering (Guangyao Shi 2020), and task assignment (Schwartz and Tokekar 2020). In those studies, however, crashed robots do not disturb correct robots as we assume in this paper. In the context of pathfinding, a few studies focus on system designs for potentially non-cooperative agents (Bnaya et al. 2013; Strawn and Ayanian 2021) where those agents can pretend to be crashed. However, those studies do not provide safe paths as presented in this paper.

**Failure Detector** The notion of a failure detector is inspired by a popular abstraction in theoretical distributed algorithms (Chandra and Toueg 1996), introduced to enable consensus solvability in an asynchronous setting. With respect to the original concept, this paper assumes the detector to be both *accurate* (i.e., it never suspects correct agents) and *complete* (i.e., it always suspects crashed agents). Removing these assumptions is an interesting future direction. Also, our use of failure detectors is *local*, that is, they provide localized information about failures, while previous works on failure detectors considered complete information oracles, that is, they provide *global* information about *all* agents.

## 8 Conclusion and Discussion

We studied a graph path planning problem for multiple agents that may crash at runtime, and block part of the workspace. Different from conventional MAPF studies, each agent can change its executing path according to local crash failure detection; hence a set of paths constitute a solution for each agent. This paper presented a safe approach to ensure that correct agents reach their destinations regardless of crash patterns, including a series of theoretical analyses. Finally, we list promising directions to extend MAPPCF.

- **Developing complete algorithms** that can address the crash awareness differences among agents.
- **Optimization:** One potential objective is minimizing the worst-case makespan (i.e., the maximum traveling time), which is convenient to practical situations.
- **Global Failure Detector:** We assumed that an agent detects crashes only when it is adjacent to crashed agents. Extending this observation range might improve the planning success rate because agents can determine their behavior based on supplementary knowledge about crashes. If the observation range is the entire graph, MAPPCF becomes a centralized problem, since every agent has the same information.

## Acknowledgments

We thank the anonymous reviewers for their many insightful comments and suggestions. This work was partly supported by JSPS KAKENHI Grant Number 20J23011, JST ACT-X Grant Number JPMJAX22A1, and ANR project SAPPORO, ref. 2019-CE25-0005-1. The work was carried out during Keisuke Okumura's staying in LIP6, partially supported by JSPS Overseas Challenge Program for Young Researchers. KO also thanks the support of the Yoshida Scholarship Foundation.

## References

Atzmon, D.; Stern, R.; Felner, A.; Wagner, G.; Barták, R.; and Zhou, N.-F. 2020. Robust multi-agent path finding and executing. *Journal of Artificial Intelligence Research (JAIR)*.

Bnaya, Z.; Stern, R.; Felner, A.; Zivan, R.; and Okamoto, S. 2013. Multi-agent path finding for self interested agents. In *Proceedings of Annual Symposium on Combinatorial Search (SOCS)*.

Čáp, M.; Novák, P.; Kleiner, A.; and Seleckỳ, M. 2015. Prioritized planning algorithms for trajectory coordination of multiple mobile robots. *IEEE Transactions on Automation Science and Engineering (T-ASE)*.

Chandra, T. D.; and Toueg, S. 1996. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*.

Felber, P.; Défago, X.; Guerraoui, R.; and Oser, P. 1999. Failure detectors as first class objects. In *Proceedings of the International Symposium on Distributed Objects and Applications*.

Guangyao Shi, L. Z., Pratap Tokekar. 2020. Robust Multiple-Path Orienteering Problem: Securing Against Adversarial Attacks. In *Proceedings of Robotics: Science and Systems (RSS)*.

Kameyama, S.; Okumura, K.; Tamura, Y.; and Défago, X. 2021. Active Modular Environment for Robot Navigation. In *Proceedings of IEEE International Conference on Robotics and Automation (ICRA)*.

Lam, E.; Le Bodic, P.; Harabor, D.; and Stuckey, P. J. 2022. Branch-and-cut-and-price for multi-agent path finding. *Computers & Operations Research (COR)*.

Okumura, K.; Bonnet, F.; Tamura, Y.; and Défago, X. 2022. Offline Time-Independent Multi-Agent Path Planning. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*.

Okumura, K.; Tamura, Y.; and Défago, X. 2021. Iterative Refinement for Real-Time Multi-Robot Path Planning. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*.

Prorok, A.; Malencia, M.; Carlone, L.; Sukhatme, G. S.; Sadler, B. M.; and Kumar, V. 2021. Beyond Robustness: A Taxonomy of Approaches towards Resilient Multi-Robot Systems. *arXiv preprint arXiv:2109.12343*.

Sartoretti, G.; Kerr, J.; Shi, Y.; Wagner, G.; Kumar, T. S.; Koenig, S.; and Choset, H. 2019. PRIMAL: Pathfinding via Reinforcement and Imitation Multi-Agent Learning. *IEEE Robotics and Automation Letters (RA-L)*.

Schwartz, R.; and Tokekar, P. 2020. Robust multi-agent task assignment in failure-prone and adversarial environments. In *Proceedings of Robotics: Science and Systems (RSS)*.

Senbaslar, B.; Hönig, W.; and Ayanian, N. 2018. Robust Trajectory Execution for Multi-robot Teams Using Distributed Real-time Replanning. In *Proceedings of International Symposium on Distributed Robotic Systems (DARS)*.

Shahar, T.; Shekhar, S.; Atzmon, D.; Saffidine, A.; Juba, B.; and Stern, R. 2021. Safe multi-agent pathfinding with time uncertainty. *Journal of Artificial Intelligence Research (JAIR)*.

Sharon, G.; Stern, R.; Felner, A.; and Sturtevant, N. R. 2015. Conflict-based search for optimal multi-agent pathfinding. *Artificial Intelligence (AIJ)*.

Sharon, G.; Stern, R.; Goldenberg, M.; and Felner, A. 2013. The increasing cost tree search for optimal multi-agent pathfinding. *Artificial Intelligence (AIJ)*.

Stern, R.; Sturtevant, N.; Felner, A.; Koenig, S.; Ma, H.; Walker, T.; Li, J.; Atzmon, D.; Cohen, L.; Kumar, T.; et al. 2019. Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks. In *Proceedings of Annual Symposium on Combinatorial Search (SOCS)*.

Strawn, K.; and Ayanian, N. 2021. Byzantine Fault Tolerant Consensus for Lifelong and Online Multi-robot Pickup and Delivery. In *Proceedings of International Symposium on Distributed Robotic Systems (DARS)*.

Surynek, P. 2019. Unifying Search-based and Compilation-based Approaches to Multi-agent Path Finding through Satisfiability Modulo Theories. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*.

Van Den Berg, J.; Guy, S. J.; Lin, M.; and Manocha, D. 2011. Reciprocal n-body collision avoidance. In *Robotics Research*.

Wiktor, A.; Scobee, D.; Messenger, S.; and Clark, C. 2014. Decentralized and complete multi-robot motion planning in confined spaces. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*.

Wurman, P. R.; D'Andrea, R.; and Mountz, M. 2008. Coordinating hundreds of cooperative, autonomous vehicles in warehouses. *AI magazine*.

Zhang, Y.; Kim, K.; and Fainekos, G. 2016. Discof: Cooperative pathfinding in distributed systems with limited sensing and communication range. In *Proceedings of International Symposium on Distributed Robotic Systems (DARS)*.

Zhou, L.; Tzoumas, V.; Pappas, G. J.; and Tokekar, P. 2018. Resilient active target tracking with multiple robots. *IEEE Robotics and Automation Letters (RA-L)*.