ECAI 2023
K. Gal et al. (Eds.)

doi:10.3233/FAIA230252

Action-Failure Resilient Planning

Diego Aineto, Alessandro Gaudenzi, Alfonso Gerevini*, Alberto Rovetta, Enrico Scala and Ivan Serina

Department of Information Engineering, University of Brescia, Italy

Abstract. In the real world, the execution of the actions planned for an agent is never guaranteed to succeed, as they can fail in a number of unexpected ways that are not explicitly captured in the planning model. Based on these observations, we introduce the task of finding plans for classical planning that are resilient to action execution failures. We refer to this problem as *Resilient Planning* and to its solutions as *K-resilient plans*; such plans guarantee that an agent will always be able to reach its goals (possibly by replanning alternative sequences of actions) as long as no more than K failures occur along the way. We also present RESPLAN, a new algorithm for Resilient Planning, and we compare its performance to methods based on compiling Resilient Planning to Fully-Observable-Non-Deterministic (FOND) planning.

1 Introduction

A solution to a classical planning problem is a plan of actions that when executed from the problem initial state is expected to reach a final state where the problem goal holds [6, 9]. Solution plans can be proven correct with respect to an abstract model of the world given in some planning formalism [14]. However, because planning is done at an abstract level, when actions do get executed in the real world they can still fail in unexpected ways. These unforeseen action failures may depend on a number of reasons, such as a deranging exogenous event, a malfunction of the necessary equipment, the lack of a resource that was assumed to be available, or an incomplete/incorrect abstract model of the action preconditions. An example is an action moving a vehicle between two connected locations across some road that, at execution time, cannot be performed because the connecting road is temporarily blocked by a car accident or some (unknown) road maintenance. Another example is a refuel action at a certain gas station that, during plan execution, we discover to be not operational (e.g., for lack of gasoline or interruption of the ATM payment service) only when we are at the gas station.

In the propositional setting of classical planning models, these kind of action failures leave the current state unaltered, since none of the effects in the model of the failed action occur. Moreover, anticipating such possible failures at planning time by means of more detailed planning models can hardly capture all possible situations in which a planned action is unexecutable, or can make the state/action models much more complex (e.g., we need to model the status of the ATM service at each gas station). On the other hand, one could reasonably expect that a rational plan-based agent incurs into action failures at most a bounded number of times along the way to reach its goal (e.g., we don't encounter many car accidents or non-operational gas stations during the same trip.)

* Corresponding Author. Email: alfonso.gerevini@unibs.it.

In automated planning, a typical way to handle action failures is interleaving plan execution and replanning from the state where a failure occurs [9], possibly by repairing the current plan instead of replanning from scratch (e.g., [1, 5, 8, 20]). However this online approach does not always guarantee that the plan under execution can be fixed achieving the original problem goal. For instance, consider a robotic domain involving path planning on a map of locations modelled by a directed graph. A valid plan going from the current location to the goal location involves executing a path of moves on such a graph connecting the source and target locations. Suppose that, at execution time, one of the plan moves is blocked by some unforeseen event or obstacle, leaving the robot at the same location/graph node. If from such a location there is no alternative path to reach the target location/graph node, then the original plan cannot be repaired.

In this paper, we propose a complementary method aimed at generating plans that have repair guarantees in case action failures will happen at execution time. We introduce the task of finding solutions to classical planning that are resilient to action failures. We refer to this problem as *Resilient Planning* and to its solution plans as *K-resilient plans*. Such plans guarantee that an agent will always be able to reach its goal (possibly by replanning online alternative sequences of actions) as long as no more than K action failures occur along the way to the goal. If more than K action failures occur when executing a K-resilient plan, any successive failure can still be handled by the execution-and-plan-repair approach but, at this point in the execution, without any guarantee that a state satisfying the problem goal can be reached.

While in general an action failure can be modeled in different ways, here we assume that it does not modify the current (abstract) state of the world. Moreover, in this work we consider stronger resilience guarantees by imposing that if an action fails, it cannot be re-applied again. We propose a new algorithm for action-failure resilient planning, called RESPLAN, that works under these assumptions. RESPLAN generates plans for classical (propositional) planning problems that are resilient to a bounded number of action execution failures. The algorithm exploits a new notion of *bounded resilient states*, and searches for solution plans that are constrained to cross only such states.

As we show in the paper, an instance of resilient planning can be reformulated as a particular instance of fully observable nondeterministic (FOND) planning [2, 3, 6]. In this compilation, the effects of any action are either the set of all its nominal effects (i.e., those of the classical planning model) or the empty set (modeling the action failure). In addition, the models of the states and action preconditions/effects are revised using additional fluents to take account of the assumption that at most K action failures can occur.

Resilient Planning resembles Fault Tolerant (FT) planning [4, 15],

which can be reformulated as another special variant of FOND Planning. FT planning models and handles action failures differently from us, and our RESPLAN algorithm substantially differs from existing techniques for FT planning.

We experimentally evaluate RESPLAN on a set of known domains for classical planning, and we compare it with a compilation-based approach using two state-of-the-art FOND planners [17, 7].

In the remainder of the paper, after a formal description of Resilient Planning, we present our algorithm and the results of the experimental evaluation. Then we discuss the related work in more detail and give the conclusions.

2 Background

A classical planning problem is a tuple $\Pi = \langle F, A, s_0, G\rangle$ whose components are defined as follows. F is a finite set of literals inducing a set S of states. A state $s \in S$ is a subset of F. If an element $f \in F$ is in a state s then f is true in s; otherwise f is false in s by the closed world assumption. s_0 is the initial state. $G \subseteq F$ is the problem goal denoting the literals that should hold in any goal state. A is a set of actions; each action $a \in A$ is specified by the pair $a = \langle \mathsf{pre}(a), \mathsf{eff}(a)\rangle$ where $\mathsf{pre}(a) \subseteq F$ is the precondition of a, and $\mathsf{eff}(a)$ the effect of a formed by subsets of positive and negative literals over F, that are denoted with $\mathsf{eff}(a)^+$ and $\mathsf{eff}(a)^-$, respectively. An action a is applicable in state s iff $\mathsf{pre}(a) \subseteq s$, and we denote the set of actions applicable in state s with $A(s)$. The application of an action $a \in A(s)$ in s generates a state $s' = s[a]$ such that, for every fluent $f \in F$, f is in s' iff $f \in (s \setminus \mathsf{eff}(a)^-) \cup \mathsf{eff}(a)^+$.

A plan π is a sequence of actions in A, i.e., $\pi = (a_1, \ldots, a_n)$. Given a planning problem $\Pi = \langle F, A, s_0, G\rangle$, $\tau = (s_0, s_1, \cdots, s_n)$ is the trajectory of states induced by applying π in s_0, i.e., $s_i = s_{i-1}[a_i]$ for $i = 1, \cdots, n$. A plan $\pi = (a_1, \ldots, a_n)$ is a solution for $\Pi = \langle F, A, s_0, G\rangle$ iff the induced trajectory of states $\tau = (s_0, s_1, \cdots, s_n)$ is such that for all $i \in [1, n]$ it holds that $\mathsf{pre}(a_i) \subseteq s_{i-1}$ and $G \subseteq s_n$.

FOND planning is an extension of classical planning where an action can have multiple alternative effect, and the state generated by its execution depends on the triggered effect. A solution to a FOND planning problem is a strong (possibly cyclic) policy that guarantees reaching the goal no matter the outcome of an action execution.

3 Planning for Resilient Solutions

Resilient planning aims at generating plans that are robust up to a given number of failures during execution. The execution of a generated plan can fail due to the abstract model used to compute the plan not fully capturing the dynamics of the domain (e.g., incorrect or incomplete action preconditions), outside interventions that impact on the possible execution of an action, or simply because an action did not have the intended outcome. Resilient planning describes the world through a (classical) planning problem but explicitly considers at planning time that actions can fail at execution time. As discussed above, we abstract the failure and recovery of actions by assuming that failures do not modify the state of the world and cannot be reapplied in the same resilient planning episode.

Our formalisation of resilient planning and its solutions rely on the following notion of *resilient states*.

Definition 1 (k-Resilient State)**.** *Let $\Pi = \langle F, A, s_0, G\rangle$ be a planning problem, S the state space induced by F, and k a non-negative integer.*

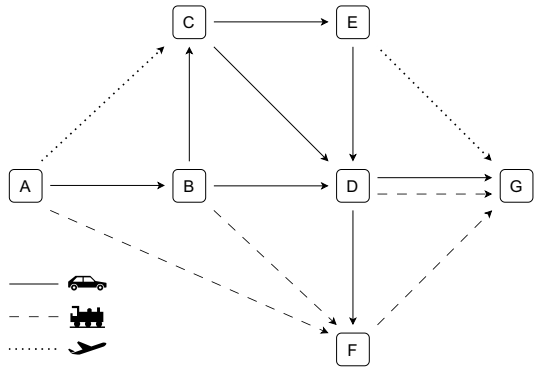

Figure 1. Navigation problem with seven locations (squares) and three types of connections: road (solid), railway (dashed), and flight (dotted).

(i) A state $s \in S$ is 0-resilient in Π iff there is a plan from s that achieves G (i.e., $\langle F, A, s, G\rangle$ is solvable);
(ii) A state $s \in S$ is k-resilient in Π if $s \models G$;
(iii) A state $s \in S$ such that $s \not\models G$ is k-resilient in Π for $k \geq 1$ iff there exists an action $a \in A(s)$ such that (1) $s[a]$ is k-resilient in Π and (2) s is $(k-1)$-resilient in $\langle F, A \setminus \{a\}, s_0, G\rangle$.

A state s is k-resilient in a planning problem $\langle F, A, s_0, G\rangle$ if the goal G can be achieved from s even after k action failures occur along the way of a plan from s to G. Def. 1 formalizes this notion by considering that the execution of an action a applicable in s can either succeed or fail. Action success is captured by Condition (1) of the case (iii), stating that the successor state $s[a]$ needs to be k-resilient. Action failure in s is captured by Condition (2) of the case (iii), requiring that s is $(k-1)$-resilient in the same planning problem but without action a, since after the failure of a we remain in state s and a should never be retried along the way from s to G. Notice that condition (iii)-(1) of Def. 1 makes the definition recursive, and case (ii) guarantees that the recursion ends in a state that is k resilient *and* entails the problem goals.

Let us illustrate the notion of a resilient state through the example of Figure 1. This figure presents a navigation problem with seven locations, representing states, and three types of connections, representing actions. The initial state of the problem is location `A` and the goal is to reach location `G`. Following Def. 1, we have that states `A`, `B`, `D` and `G` are 2-resilient, states `C` and `E` are 1-resilient, and state `F` is 0-resilient. To see this we can reason backwards starting from the goal state `G`, which is resilient for any value of k by definition. For example, consider state `F`. We can use `train(F,G)` to move to `G` but, if this action fails, there is no other way to reach the goal which makes `F` a 0-resilient state. Next, let us consider state `D`. This state is 2-resilient because we can move from `D` to `G` using `car(D,G)`, if `car(D,G)` fails we can use `train(D,G)` and finally, if both `car(D,G)` and `train(D,G)` fail, we still can travel through `F`.

We can see that the definition of k-resilient state implies that there exists a trajectory from the state to the goal where all states are k-resilient. This observation takes us to our next definition.

Definition 2 (k-Resilient Plan)**.** *Given a planning problem $\Pi = \langle F, A, s_0, G\rangle$, a solution plan for Π that induces a state trajectory $(s_0, s_1, \ldots, s_n)$ is k-resilient for Π if, for all $0 \leq i < n$, it holds that s_i is k-resilient in Π.*

By the previous definitions, a k-resilient plan π has the property that, *if during the execution of π any action a fails in a state s, it will*

still be possible to achieve the problem goal from s by an alternative plan π' that does not use a and is resilient to $k-1$ failures. In our example of Figure 1, we have two 2-resilient plans. Both plans follow the path A-B-D-G and only differ in whether we use `car(D,G)` or `train(D,G)` to move from D to G. There are also several 1-resilient plans like (`plane(A,C)`, `car(C,E)`, `plane(E,G)`) and (`car(A,B)`, `car(B,C)`, `car(C,D)`, `train(D,G)`) and, of course, any path from A to G corresponds to a 0-resilient plan.

Definition 3 (Resilient Planning)**.** *Given a planning problem Π and an integer $K \geq 0$, Resilient Planning is the computational problem of finding a K-resilient plan for Π, if one exists, and returning "unsolvable" otherwise.*

A *resilient planning problem* is a pair $\langle \Pi, K \rangle$ where Π is a planning problem and K is an non-negative integer (classical planning is a special case of resilient planning with $K = 0$). A *solution* for $\langle \Pi, K \rangle$ is a K-resilient plan for Π.

The next theorem states a property about the resilience of the problem initial state that is exploited by our planning algorithm presented in the next section.

Theorem 1. *Let $\langle \langle F, A, s_0, G \rangle, K \rangle$ be a resilient planning problem. State s_0 is K-resilient for $\langle F, A, s_0, G \rangle$ if and only if there exists a solution for $\langle \langle F, A, s_0, G \rangle, K \rangle$.*

Proof. By Definitions 2-3, if $\langle \langle F, A, s_0, G \rangle, K \rangle$ is solvable, then s_0 must be K-resilient for $\langle F, A, s_0, G \rangle$. Def. 1 guarantees that s_0 is k-resilient only if there exists a plan from s_0 that reaches the goal G and generates a trajectory of states that are all k resilient, i.e., only if $\langle \langle F, A, s_0, G \rangle, K \rangle$ has a solution. □

4 An Algorithm for Resilient Planning

This section proposes a novel algorithm for Resilient Planning called RESPLAN. Intuitively, RESPLAN computes resilient plans for classical planning problems by iteratively using a classical planner to prove whether the initial state of the problem is resilient or not.

4.1 *The* RESPLAN *Algorithm*

RESPLAN takes as input a resilient planning problem $\langle \langle F, A, s_0, G \rangle, K \rangle$, and outputs a K-resilient plan π^K for $\langle F, A, s_0, G \rangle$ if it exists and *unsolvable* otherwise. RESPLAN leverages the theoretical result of Theorem 1 to pose the problem of finding the solution K-resilient plan as the problem of proving that the initial state s_0 is K-resilient. The recursive nature of the definition of k-resilient state means that, in order to achieve this, we will have to prove the resilience of many other states. That is, to prove that a state s is k-resilient, we need to find a successor state $s' = s[a]$ that is also k-resilient, and we also need to show that s is still $(k-1)$-resilient without using action a. The RESPLAN algorithm does this by performing a search for resilient states over an augmented state space that combines to the classical state with the number of failures detected so far and the faulty actions. Nodes in this augmented space are tuples of the form $\langle s, k, V \rangle$ where s is a state, $0 \leq k \leq K$, and $V \subseteq A$ are faulty actions that cannot be used again. Implicitly, a node $\langle s, k, V \rangle$ represents the problem of deciding whether s is a k-resilient state in $\langle F, A \setminus V, s_0, G \rangle$.

The pseudocode of RESPLAN is reported in Algorithm 1, which we describe in the following. RESPLAN maintains a Last-In-First-Out (LIFO) list $Open$ and two sets, $\mathcal{R}_\uparrow$ and $\mathcal{R}_\downarrow$, all containing nodes $\langle s, k, V \rangle$. The $Open$ list stores all nodes that still need to be evaluated. Each of these nodes will be either moved to $\mathcal{R}_\uparrow$, if proven resilient, or to $\mathcal{R}_\downarrow$ if proven non-resilient. Notice that $\mathcal{R}_\uparrow$ and $\mathcal{R}_\downarrow$ are complementary sets that can be understood as a sort of closed list in our algorithm. Since the aim is to prove the initial state K-resilient, we initialize $Open$ with $\langle s_0, K, \emptyset \rangle$.

Main loop (lines 2-20): In each iteration, the main loop pops a node $\langle s, k, V \rangle$ from $Open$ (line 3) and determines whether it belongs to either $\mathcal{R}_\uparrow$ or $\mathcal{R}_\downarrow$ already. RESPLAN first checks if the node can already be proven k-resilient by calling the $RCheck$ function which verifies that case (iii) of Def. 1 is satisfied. If $RCheck$ returns True, the node is added to $\mathcal{R}_\uparrow$ (line 6). Otherwise, RESPLAN needs to find additional resilient states (through which we can reach the goal) in order to prove that s is k-resilient. This is done by generating a plan-

Algorithm 1 RESPLAN

Input Resilient Planning problem $\langle \Pi = \langle F, A, s_0, G \rangle, K \rangle$
Output K-resilient plan π^K if it exists; unsolvable, otherwise

```
1:  Open := {⟨s_0, K, ∅⟩}; R↑ := ∅; R↓ := ∅;
2:  while Open ≠ ∅ do
3:      ⟨s, k, V⟩ := Open.pop()
4:      if ⟨s, k, V⟩ ∉ R↑ ∪ R↓ then
5:          if RCheck(s, k, V, A, G, R↑) then
6:              R↑.add(s, k, V)
7:          else
8:              Π' := ⟨F, A \ V, s, G⟩
9:              S↓ := {s' | ⟨s', k, V⟩ ∈ R↓}
10:             π, τ := ComputePlan(Π', S↓)
11:             if π = null then
12:                 UpdateNonResilient(s, k, V, R↓)
13:             else if k ≥ 1 then
14:                 for i = 1 to i = |π| do
15:                     Open.push(τ_{i-1}, k, V)
16:                     Open.push(τ_{i-1}, k − 1, V ∪ {π_i})
17:                 R↑.add(τ_{|τ|}, k, V)
18:             else
19:                 for i = 1 to i = |π| + 1 do
20:                     R↑.add(τ_{i-1}, 0, V)
21: if ⟨s_0, K, ∅⟩ ∈ R↑ then
22:     π^K := ExtractSolution(Π, K, R↑)
23:     return π^K
24: else
25:     return unsolvable
26:
27: function RCheck(s, k, V, A, R↑)
28:     for a ∈ A(s) \ V do
29:         if (⟨s[a], k, V⟩ ∈ R↑ ∧ ⟨s, k − 1, V ∪ {a}⟩ ∈ R↑ then
30:             return True
31:     return False
32:
33: function ExtractSolution(⟨F, A, s_0, G⟩, K, R↑)
34:     s := s_0
35:     π^K := ()
36:     while s ⊭ G do
37:         for a ∈ A(s) do
38:             if ⟨s[a], K, ∅⟩ ∈ R↑ then
39:                 s := s[a]
40:                 π^K.append(a)
41:                 break
42:     return π^K
```

ning problem $\Pi' = \langle F, A \setminus V, s, G\rangle$ and calling $ComputePlan$ to solve it. The $ComputePlan$ function encapsulates a classical planner and returns a plan π and the corresponding induced state trajectory τ that solves Π' without visiting any state that is already known to *not* be k-resilient in Π'. There are three possible outcomes at this point. First (lines 11–12), no plan is returned by $ComputePlan$, which means that s is not k-resilient in $\langle F, A \setminus V, s_0, G\rangle$. When this happens, we call the $UpdateNonResilient$ procedure to add $\langle s, k, V\rangle$ to $\mathcal{R}_\downarrow$. The second possibility, lines 13 to 17, is that $ComputePlan$ did return a plan and $k \geq 1$ so, following case (iii) of Def. 1, every state traversed by τ needs to be proven k-resilient as well as $(k-1)$-resilient without the executed action. This is captured in the algorithm by pushing new nodes into the $Open$ list in lines 15 and 16. Note that we push first node $\langle \tau_{i-1}, k, V\rangle$ and then node $\langle \tau_{i-1}, k-1, V \cup \pi_i\rangle$ from the beginning to the end of trajectory τ, and recall that $Open$ is structured as an LIFO list. This means that RESPLAN starts proving the resilience of the generated states from the back of the plan and for lower values of k first, implementing a sort of depth-first search. The last case is when $k = 0$ (lines 18–20), so finding a plan already proved every state in τ to be 0-resilient; consequently, we can add every state in τ to the $\mathcal{R}_\uparrow$. The algorithm terminates once $Open$ is empty, at which point $\langle s_0, K, \emptyset\rangle$ will either be in $\mathcal{R}_\uparrow$, if a K-resilient plan exists, or in $\mathcal{R}_\downarrow$, otherwise. Note that $\langle s_0, K, \emptyset\rangle$ will always occupy the first position in $Open$ and, therefore, be the last one to be popped. It is worth noticing that, at line 15, we are pushing a state from which we already have a plan to reach the goal. This node will eventually be popped from the open list again and if the $RCheck$ returns True, this state will be deemed resilient. Otherwise, the algorithm will attempt a new plan from this state.

Handling non-resilient states: As said above, a non-resilient state $\langle s, k, V\rangle$ is identified when no plan is returned by the function $ComputePlan$, which means that we have exhausted all possible ways to prove the k-resilience of s in Π' without succeeding. What we omitted to say is that just adding $\langle s, k, V\rangle$ to $\mathcal{R}_\downarrow$ is not enough to prevent the node $\langle s, k, V\rangle$ from being generated again. Note that the set $S_\downarrow$ that RESPLAN generates in line 9 contains the states that are known to not be k-resilient and will prevent the algorithm from pushing them into $Open$ in line 15 (the plans generated at line 10 cannot cross such states). On the other hand, if we are in an iteration where the popped node is $\langle s', k+1, V'\rangle$ with $V' \subset V$ ($|V'| = |V|-1$) and the plan from s' returned by $ComputePlan$ visits s, node $\langle s, k, V\rangle$ will be pushed again into the $Open$ list in line 16. To avoid this situation, we exploit that it is possible to propagate non-resilient states found for lower values of resilience to higher values. This is formalised by the following proposition derived from Def. 1.

Proposition 1. *Let $\Pi = \langle F, A, s_0, G\rangle$ be a planning problem, V a subset of A, and s a state of Π. If s is not k-resilient in $\langle F, A \setminus V, s_0, G\rangle$, then s is also not k'-resilient in $\langle F, A \setminus V', s_0, G\rangle$ for any $V' \subseteq V$ and $k' = k + |V \setminus V'|$.*

This proposition says that a state that is not k-resilient will never become $(k+n)$-resilient by allowing n more actions in the planning problem. To understand this let us revisit our example of Figure 1. The state `F` is 0-resilient but not 1-resilient. Proposition 1 states that if we add another action, say that we add `car(F,G)` through a new road, the state `F` will not be 2-resilient (although it becomes 1-resilient) and generalizes this idea for any number of added actions.

Going back to our algorithm, we apply this result in procedure $UpdateNonResilient$ to update $\mathcal{R}_\downarrow$ with $\langle s, k, V\rangle$ and all other non-resilient states that can be derived by Proposition 1. In this way, given a node $\langle s, k, V\rangle$, this procedure will add to $\mathcal{R}_\downarrow$ all nodes $\langle s, k', V'\rangle$ where $V' \subseteq V$ and $k' = K - |V'|$. Note also that the number of added nodes is always $2^{|V|}$. Ultimately, by leveraging this proposition, we prevent RESPLAN from pushing $\langle s, k, V\rangle$ into $Open$ again if it has been previously proven that s is not k-resilient because $S_\downarrow$ will contain s.

Extracting a solution: The RESPLAN algorithm does not store explicitly the solution K-resilient plan. Instead, the solution can be computed from $\mathcal{R}_\uparrow$ following a simple procedure depicted in the $ExtractSolution$ function. To extract the K-resilient plan we can greedily take any action $a \in A(s)$ (starting with $s = s_0$) that takes us to another K-resilient state until we reach the goal.

4.2 *Example of* RESPLAN *Execution*

Now that we have explained the algorithm, let us give a quick walk-through of what could be a possible execution of RESPLAN. We consider as inputs the problem of Figure 1 and $K = 2$, so the $Open$ list will be initialized with $\langle \mathtt{A}, 2, \emptyset\rangle$. Assume that, in the first iteration, $ComputePlan$ returns the plan `(car(A,B),train(B,F),train(F,G))` so, after pushing the generated nodes, $\langle \mathtt{F}, 1, \{\mathtt{train(F,G)}\}\rangle$ will occupy the last position in $Open$. In the next iteration, RESPLAN will pop $\langle \mathtt{F}, 1, \{\mathtt{train(F,G)}\}\rangle$ and it will find out that `F` is not 1-resilient, since $ComputePlan$ will not be able to find a plan from `F` to `G` that does not use `train(F,G)`. It will then call $UpdateNonResilient$ and add $\langle \mathtt{F}, 1, \{\mathtt{train(F,G)}\}\rangle$ to $\mathcal{R}_\downarrow$ alongside $\langle \mathtt{F}, 2, \emptyset\rangle$, which is derived from Proposition 1. At this point, the plan `(car(A,B),train(B,F),train(F,G))` is not 2-resilient since `F` is not 2-resilient, and this will be reflected in the algorithm by failing the $RCheck$ after it pops $\langle \mathtt{B}, 2, \emptyset\rangle$. RESPLAN will then try to compute a plan to the goal starting from `B`. Recall that $\langle \mathtt{F}, 2, \emptyset\rangle$ belongs to $\mathcal{R}_\downarrow$ and, therefore, $S_\downarrow$ contains `F` which will be in turn not visited by $ComputePlan$. Let us assume that the computed plan is `(car(B,D),car(D,G))` It is worth noticing that, at this stage, the algorithm discarded the suffix `(train(B,F),train(F,G))` of the first computed plan `(car(A,B),train(B,F),train(F,G))`, and is now considering `(car(A,B),car(B,D),car(D,G))`. Next, RESPLAN will pop $\langle \mathtt{D}, 1, \{\mathtt{car(D,G)}\}\rangle$ and compute a plan from `D` to `G` without using `car(D,G)`. One possibility is to use `train(D,G)` instead, so, in the next iteration, it will pop $\langle \mathtt{D}, 0, \{\mathtt{car(D,G)}, \mathtt{train(D,G)}\}\rangle$. At this point it will once again compute a plan from `D` to `G`, but this time without using neither `car(D,G)` nor `train(D,G)`, and the only solution here will be `(car(D,F),train(F,G))`. Since we are at $k = 0$, the nodes $\langle \mathtt{D}, 0, \{\mathtt{car(D,G)}, \mathtt{train(D,G)}\}\rangle$ and $\langle \mathtt{F}, 0, \{\mathtt{car(D,G)}, \mathtt{train(D,G)}\}\rangle$ will be directly added to $\mathcal{R}_\uparrow$. In the next two iteration, RESPLAN will pop first $\langle \mathtt{D}, 1, \{\mathtt{car(D,G)}\}\rangle$ and then $\langle \mathtt{D}, 2, \emptyset\}\rangle$, and both times the $RCheck$ will return True; so these nodes will be moved to $\mathcal{R}_\uparrow$. At this point, RESPLAN has proven that `D` is 2-resilient. The following iterations, all the way to the end of the algorithm, will follow a pattern similar to the one described for `D` but for states `B` and `A`. Once RESPLAN terminates, it will return the 2-resilient plan `(car(A,B),car(B,D),car(D,G))`.

4.3 *Theoretical Properties*

This section discusses the theoretical properties of RESPLAN. Note that in proving these properties we assume that the classical planner used by the $ComputePlan$ function is both sound and complete.

Theorem 2. *Given a Resilient Planning problem* $\langle\langle F, A, s_0, G\rangle, K\rangle$*,* RESPLAN *returns a solution iff* $\langle F, A, s_0, G\rangle$ *is solvable. Otherwise, it returns "unsolvable".*

Proof sketch. First, we show that the algorithm always terminates in a finite number of steps. Indeed, the set of nodes $\langle s, k, V\rangle$ that can be pushed into $Open$ is finite as all three components can take a finite number of different values, and for any node $\langle s, k, V\rangle$ that is added to the $Open$ list, RESPLAN will eventually prove or disprove the k-resilience of s. That is, it cannot remain unknown. To see why this is true, observe that when we pop a node from $Open$, this node will be either (1) added to the $\mathcal{R}_\uparrow$ set (if function $RCheck$ returns true), (2) added to the $\mathcal{R}_\downarrow$ set in line 12, or (3) pushed again into the $Open$ list in the first iteration of line 15, after having computed a plan from s avoiding states $S_\downarrow$ through the $ComputePlan$ function. Case (3) happens when RESPLAN searches for alternative plans because some state traversed by the previously generated plan from s was found to be non-resilient. Every time this happens $\mathcal{R}_\downarrow$ is then populated with new nodes, which are a finite number. It follows that the same node can re-enter into $Open$ a finite number of times, with the worst case when $\mathcal{R}_\downarrow$ eventually saturates. At that point, no solution will be found by $ComputePlan$, and the node will be moved to $\mathcal{R}_\downarrow$.

When the algorithm finds a solution, it means that it has proved that the initial state is K-resilient. Then, to ensure the correctness of RESPLAN for solvable instances, it suffices to show that for every node $\langle s, k, V\rangle$ in $\mathcal{R}_\uparrow$ it holds that s is k-resilient in $\langle F, A \setminus V, s_0, G\rangle$. It is easy to see that this is true since the set $\mathcal{R}_\uparrow$ is only populated in lines 6, 17, and 20, which correspond, respectively, to cases (iii), (ii) and (i) of Def. 1.

We also show that $\mathcal{R}_\downarrow$ only contains nodes associated to non-resilient states. By contradiction, assume that $\langle s, k, V\rangle$ belongs to $\mathcal{R}_\downarrow$ and s is k-resilient in $\langle F, A \setminus V, s_0, G\rangle$. Observe that $\mathcal{R}_\downarrow$ is only populated in line 12 after $ComputePlan$ returns no solution. There are two possibilities then. First, $\langle F, A \setminus V, s_0, G\rangle$ cannot be solved without visiting a non-resilient state, which contradicts the assumption that s is k-resilient. The second possibility is that it was added following the generalization of non-resilient states of Proposition 1, which would contradict the proposition itself.

Finally, completeness is proved by observing that RESPLAN never computes the same plan at line 10 each time the same node is popped from $Open$, and that by construction of $R_\downarrow$ only plans that visit non-resilient states are pruned (and such plans cannot be solutions). □

5 Experimental Evaluation

In this section, we analyse the performance of the presented RESPLAN algorithm to solve Resilient Planning problems. We implemented RESPLAN starting from the available code of PRP [18] and used FastDownward [10] with the h_{FF} heuristic [12] as the classical planner of $ComputePlan$. The code of the algorithm and the benchmarks are publicly available[1].

5.1 Experimental setup

We evaluated our algorithm on two sets of problems created by taking classical planning instances and scaling the value of K from 1 to 4. For the first set, we took a selection benchmarks from the previous and past International Planning Competitions (IPC) where we can expect to find some resilient solutions. These are domains that display some redundant components; this is what allows problems to have alternative ways to achieve the goals. Transportation domains are good candidates since they usually have several vehicles that can deliver and load, and navigation graphs that allow different paths to move among locations. With this in mind, the domains we selected were Driverlog, Satellite, Storage and Zenotravel. These four domains, while similar, present different flavours that are interesting for resilience. In Driverlog it is necessary to secure drivers and the navigation graph is not fully-connected, Satellites are equipped with different (possibly overlapping) collections of instruments required for the tasks, Storage introduces hoists to move crates around, and in Zenotravel planes can travel at two speeds and consume fuel.

In addition, we also created a second benchmark set where we have manually generated problems to challenge the approaches to look for solutions for high levels of resilience. The domains that make up this benchmark are BlocksworldMA, Rockets and Logistics. BlocksworldMA is the multi-agent variant of the classical Blocksworld domain that introduces several arms, Rockets is a simple transportation domain where resilience is strictly related to the number of rockets, and Logistics is a more involved domain that considers several cities, each with their own locations and vehicles. For each domain, we generated 10 instances by introducing an appropriate number of objects of each type that allows for high resilience solutions, and increasing their difficulty by considering more goals.

As baselines for the evaluation, we considered the specialized algorithms of [15] and a compilation to FOND planning inspired by this same work. However, we focus on the compilation approach as the specialized algorithms only handle problems with K=1, and could not solve even the smallest instance in our benchmarks suite. In the FOND planning problem, we enforce our semantics in the action models, namely, 1) all actions can fail and the failure does not modify the state, and 2) actions that have failed cannot be used again. Assumption (1) can be encoded by extending all actions with a new secondary empty effect. Assumption (2) requires extending the actions preconditions and the new secondary effect with a proposition $f_a^\checkmark$ that denotes that action $a \in A$ has already failed. As done in [15], we limited the number of faults by introducing a counter in the problem. This counter disables secondary effects once the bound K is reached. To solve the compiled problems, we use MyND (with the h_{FF} heuristic) and FOND-SAT, two FOND planners capable of computing strong policies. All experiments were run on a Xeon Gold 6140M at 2.3 GHz, with a time limit of 1800s and a memory limit of 32GB for each problem.

5.2 Results

In the following, we discuss the obtained results. We start our commentary by looking at the coverage results over our two sets of benchmark problems and then move on to the runtime analysis.

Coverage results: Table 1 summarizes the coverage results. We have aggregated separately the coverage score for solvable (denoted by an "S") and unsolvable instances (denoted by a "U") to analyze these two cases more in depth. Columns MN and FS correspond to the results of the compilation solved by MyND and FOND-SAT planners, respectively, while RP refers to the results of our algorithm. We note that FOND-SAT, as a SAT-based planning algorithm, is unable to prove unsolvability and remark this by using a "-" in the corresponding table entries. Focusing first on the IPC benchmark, we observe that K-resilient plans are only found for $K \leq 2$ and problems become unsolvable as we increase K. We can see that RESPLAN outperforms the baselines in solvable instances. In particular, RESPLAN is able to find 56 solutions for $K = 1$ and 12 for $K = 2$, while

[1] https://github.com/ale-gaudenzi/resilient-planner

		$K=1$			$K=2$			$K=3$			$K=4$		
Domain	**Sol**	**MN**	**FS**	**RP**	**MN**	**FS**	**RP**	**MN**	**FS**	**RP**	**MN**	**FS**	**RP**
Driverlog	S	2	1	14	0	0	1	0	0	0	0	0	0
(#20)	U	0	-	0	0	-	0	0	-	0	1	-	0
Satellite	S	1	0	17	1	0	5	0	0	0	0	0	0
(#36)	U	1	-	1	0	-	1	0	-	1	0	-	1
Storage	S	4	0	10	0	0	3	0	0	0	0	0	0
(#30)	U	6	-	5	9	-	7	9	-	5	12	-	6
Zenotravel	S	3	1	15	1	1	3	0	0	0	0	0	0
(#20)	U	1	-	1	0	-	1	1	-	2	2	-	2
Subtotal-IPC	S	10	2	**56**	2	1	**12**	0	0	0	0	0	0
(#106)	U	**8**	-	7	**9**	-	**9**	**10**	-	8	**15**	-	9
BlocksworldMA	S	5	6	10	0	2	10	0	0	5	0	0	4
(#10)	U	0	-	0	0	-	0	0	-	0	0	-	0
Logistics	S	0	1	9	0	0	9	0	0	7	0	0	3
(#10)	U	0	-	0	0	-	0	0	-	0	0	-	0
Rocket	S	4	4	10	1	0	10	0	0	9	0	0	2
(#10)	U	0	-	0	0	-	0	0	-	0	0	-	0
Subtotal-Res	S	9	11	**29**	1	2	**29**	0	0	**21**	0	0	**9**
(#30)	U	0	-	0	0	-	0	0	-	0	0	-	0
Total	S	19	13	**85**	3	3	**41**	0	0	**21**	0	0	**9**
(#136)	U	**8**	-	7	**9**	-	**9**	**10**	-	8	**15**	-	9

Table 1. Coverage results for RESPLAN (RP), compilation into FOND using MyND (MN) or FOND-SAT (FS) across benchmarks from the IPCs and the newly generated instances (Res)

MyND only finds 10 and 2 solutions, repectively. FOND-SAT performs quite poorly in this benchmarks, only finding 2 solutions for $K = 1$ and 1 for $K = 2$. The results for unsolvable instances are more mixed and, interestingly, seem to indicate that MyND scales better with the value of K for these cases. We attribute this to the way RESPLAN structures the search in depth-first fashion since, for high values of K, it may run into situations where it spends the majority of the effort in proving the resilience of a suffix of a plan that is later found to not be K-resilient.

Moving now to the second benchmark set, we can observe that we have been able to find that most instances have 3-resilient plans and some even for 4-resilient plans. RESPLAN finds solutions to 29 out of the 30 instances for $K = 1$ and $K = 2$, while the coverage of the baselines is significantly lower. Even for $K = 3$ and $K = 4$ where both MyND and FOND-SAT failed to find any solution, RESPLAN is able to find 21 and 9 solutions, respectively. The results here confirm the efficiency of RESPLAN for solvable instances as it is able to find solutions for very large planning problems where the other baselines struggled. Interestingly, neither RESPLAN nor MyND are able to prove unsolvability in this benchmark, possibly because the state-space is too large in these instances. Overall, the results show that RESPLAN dominates the baselines for solvable instances and performs comparatively well in unsolvable ones. Nevertheless, these results serve as a stimulus to explore novel pruning techniques that help detect unsolvable instances faster.

Runtime results: We now evaluate the runtime of our algorithm. Figure 2 presents a runtime comparison between RESPLAN and the compilation solved with MyND across all benchmarks and resilience values; we exclude FOND-SAT from this analysis as there are not enough data points due to its low overall coverage. We can see that there are many instances where RESPLAN is able to quickly solve the problem while MyND runs out of time. This happens often for solvable instances, as we saw earlier, and highlights the efficiency of RESPLAN to find K-resilient plans. More interestingly, the results clearly indicate that the compared approaches are quite complementary, with MyND performing generally better than RESPLAN in proving unsovability.

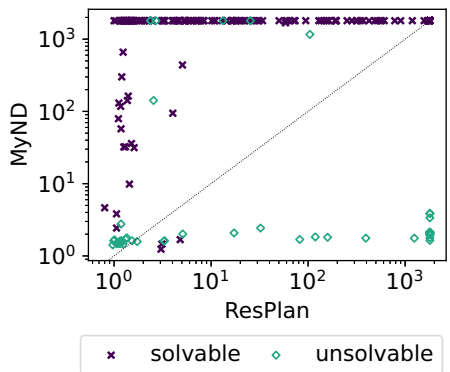

Figure 2. RESPLAN (x-axis) vs compilation solved by MyND (y-axis). Points represent runtime in seconds over all instances.

Figure 3 shows the cumulative number of instances solved over time for RESPLAN and the two baselines. We can observe that RESPLAN is able to solve more instances than MyND and FOND-SAT across all values of K. RESPLAN completely dominates for $K = 1$ and $K = 2$; in particular, RESPLAN with $K = 1$ is extremely fast and is able to solve 63 instances (over 85 instances solved) in less than 10 seconds, while solving with $K = 2$ requires much more time (as expected given the high number of additional states to examine) but is still very efficient. For $K = 3$ and $K = 4$, we see that MyND is able to solve some instances (by proving them unsolvable) in a few seconds, but after that it fails to obtain any new solution and is surpassed by RESPLAN.

6 Related Work

Resilient planning is related to planning under uncertainty, a topic of automated planning that has been tackled from a variety of perspectives [6]. Here we are interested in the case that deals with uncertainty in the sense of non-deterministic non-stochastic behaviors,

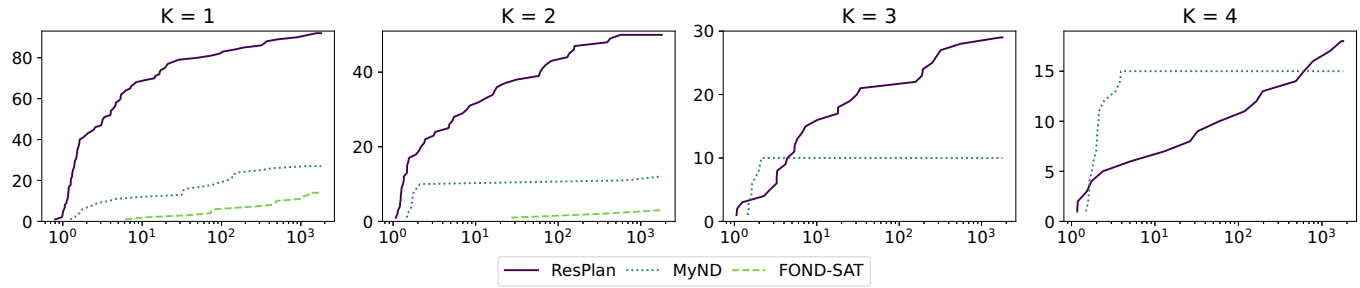

Figure 3. Coverage (y-axis) vs runtime (x-axis) for increasing values of K.

where a given agent has to anticipate unexpected contingencies at planning at time, but assume to have full observability of the world once she applies actions. This is generally referred to as FOND for Full-Observable-Non-Deterministic planning, which in its general formulation is an EXPTIME-complete problem [16]. FOND problems have been approached via replanning (e.g., [21, 18]), symbolic methods (e.g., [2, 7]) or native methods (e.g., [11], [19], [17]).

Resilient Planning (RP for short), can be formulated as a specific restriction of FOND planning that has bounded indeterminacy in which we assume that the agent action can fail producing none of the modeled effects, but failures can happen only a limited number of times. This restriction is similar to Fault-Tolerant-Planning (FT), a fragment of FOND planning initially studied in [15]. In FT, the action model devises primary and secondary effects. Secondary effects are those that happen when some unexpected situation which is not under the control of the agent occurs. The main objective of FT planning is to generate plans guaranteed to reach a goal state as long as no more than a given number k of secondary effects are triggered during execution. FT planning was later generalized and studied in [4] from a computational complexity standpoint.

The work in [15] proposes to solve FT problems by an OBDD based algorithm supplied in different configurations: a blind backward search and more specialised algorithms for the case where $k = 1$. The specialised versions can use different variants of heuristics based on the syntactic structure of the states investigated by the OBDD representation. The work in [4] focuses on the theoretical aspects of an extended version of FT having m primary effects; for $m = 1$, which is the case studied in [15], FT is proved to be PSPACE-complete and therefore easier than FOND planning (assuming PSPACE$\neq$EXPTIME).

We observe that a RP problem can be cast as a FT problem with one secondary effect. Such a secondary effect formalizes the contingency when the action simply leaves the state unaltered, which in our semantics corresponds to an action failure. Moreover, we need to be sure that the failing action cannot be re-applied (in the same or successive states); this can be enforced by making such an action inapplicable. Because of this, FT can be seen as a more expressive formalism than RP. Yet, we argue that RP provides a complementary approach to FT in that it allows to model failures in more abstract terms, and the user is not asked to formulate a non-deterministic model for each action. Indeed, in RP the non-determinism is implicit in the semantics of the problem itself.

From the point of view of the solution algorithms, our RESPLAN substantially differs from what is presented in [15] for FT, as well from algorithms for FOND planners. In comparison with the algorithm proposed by [15], our algorithm has no restrictions on the input number of allowed failures, and exploits classical planners almost off-the-shelf, leveraging the vast amount of work done in devising informed heuristics from classical planning.

PRP [18] is a state of the art planner for FOND planning that has some similarity with RESPLAN. PRP works by iteratively calling a classical planner to build a strong cyclic policy for the given FOND planning problem. To improve its performance, PRP collects and exploits dead-ends during search so that the exploration of useless states is avoided by the classical planner problem formulated on the fly. Differently from PRP, RP requires solutions to be strong *non*-cyclic. This is because in our setting there is no fairness assumption as in PRP, and therefore we do not assume the case that the nominal effect of an action will occur in the limit if it is repeatedly applied.

7 Conclusion

We have addressed the problem of generating plans in the context of classical planning that are robust during execution. Such plans are required to cross only states that satisfy a property of bounded resilience introduced in the paper. Our RESPLAN algorithm generates resilient plans that are guaranteed to be repairable when at most a bounded number of planned actions cannot be executed or provide no effect, leaving the current state of the planning model unaltered.

An experimental comparison with an alternative approach based on compilation into strong FOND planning indicates that, RESPLAN is much more effective in terms of both coverage and run-time over solvable instances, and it is competitive in terms of coverage over unsolvable instances.

In future work, we plan to optimise the performance of RESPLAN in different ways, and in particular by novel pruning techniques that can make RESPLAN more efficient for unsolvable instances, for instance by exploiting landmarks [13]. Moreover, we intend to study alternative notions of resilient states and further ways to model and handle action failures.

Finally, an interesting possible use of resilient planning that we have not investigated in this paper concerns the formalisation of the planning problem. A real-world planning problem could be abstractly represented in classical planning by different alternative models (actions with different preconditions and effects, and states with different fluents). Resilient planning could be a tool to evaluate the quality of such models in terms of the degree of resilience they admit in their plans, preferring models that support higher resilience. This is another direction for further research.

Acknowledgements

This research has been carried out with the support of EU H2020 project AIPlan4EU (GA n. 101016442), EU ICT-48 2020 project TAILOR (No. 952215), and MUR PRIN project RIPER (No. 20203FFYLK).

References

[1] Mohannad Babli, Óscar Sapena, and Eva Onaindia, 'Plan commitment: Replanning versus plan repair', *Engineering Applications of Artificial Intelligence*, **123**, (2023).
[2] Alessandro Cimatti, Marco Pistore, Marco Roveri, and Paolo Traverso, 'Weak, strong, and strong cyclic planning via symbolic model checking', *Artif. Intell.*, **147**(1-2), 35–84, (2003).
[3] Alessandro Cimatti, Marco Roveri, and Paolo Traverso, 'Automatic obdd-based generation of universal plans in non-deterministic domains', in *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, pp. 875–881. AAAI Press / The MIT Press, (1998).
[4] Carmel Domshlak, 'Fault tolerant planning: Complexity and compilation', in *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 23, pp. 64–72, (2013).
[5] Maria Fox, Alfonso Gerevini, Derek Long, and Ivan Serina, 'Plan stability: Replanning versus plan repair', in *Proceedings of the Sixteenth International Conference on Automated Planning and Scheduling (ICAPS-2006)*, pp. 212–221. AAAI, (2006).
[6] Hector Geffner and Blai Bonet, *A Concise Introduction to Models and Methods for Automated Planning*, Synthesis Lectures on Artificial Intelligence and Machine Learning, Morgan & Claypool Publishers, 2013.
[7] Tomas Geffner and Hector Geffner, 'Compact policies for fully observable non-deterministic planning as SAT', in *ICAPS*, pp. 88–96. AAAI Press, (2018).
[8] Alfonso Gerevini and Ivan Serina, 'Efficient plan adaptation through replanning windows and heuristic goals', *Fundamenta Informaticae*, **102**(3-4), 287–323, (2010).
[9] Malik Ghallab, Dana S. Nau, and Paolo Traverso, *Automated Planning and Acting*, Cambridge University Press, 2016.
[10] Malte Helmert, 'The fast downward planning system', *Journal of Artificial Intelligence Research*, **26**, 191–246, (2006).
[11] Jörg Hoffmann and Ronen I. Brafman, 'Contingent planning via heuristic forward search witn implicit belief states', in *ICAPS*, pp. 71–80. AAAI, (2005).
[12] Jörg Hoffmann and Bernhard Nebel, 'The ff planning system: Fast plan generation through heuristic search', *Journal of Artificial Intelligence Research*, **14**, 253–302, (2001).
[13] Jörg Hoffmann, Julie Porteous, and Laura Sebastia, 'Ordered landmarks in planning', *J. Artif. Intell. Res.*, **22**, 215–278, (2004).
[14] Richard Howey, Derek Long, and Maria Fox, 'Val: Automatic plan validation, continuous effects and mixed initiative planning using pddl', in *16th IEEE International Conference on Tools with Artificial Intelligence*, pp. 294–301. IEEE, (2004).
[15] Rune M Jensen, Manuela M Veloso, and Randal E Bryant, 'Fault tolerant planning: Toward probabilistic uncertainty models in symbolic non-deterministic planning.', in *ICAPS*, pp. 335–344, (2004).
[16] Michael L. Littman, 'Probabilistic propositional planning: Representations and complexity', in *AAAI/IAAI*, pp. 748–754. AAAI Press / The MIT Press, (1997).
[17] Robert Mattmüller, Manuela Ortlieb, Malte Helmert, and Pascal Bercher, 'Pattern database heuristics for fully observable nondeterministic planning', in *ICAPS*, pp. 105–112. AAAI, (2010).
[18] Christian J. Muise, Sheila A. McIlraith, and J. Christopher Beck, 'Improved non-deterministic planning by exploiting state relevance', in *ICAPS*. AAAI, (2012).
[19] Ramon Fraga Pereira, André Grahl Pereira, Frederico Messa, and Giuseppe De Giacomo, 'Iterative depth-first search for FOND planning', in *ICAPS*, pp. 90–99. AAAI Press, (2022).
[20] Sung Wook Yoon, Alan Fern, and Robert Givan, 'Ff-replan: A baseline for probabilistic planning', in *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling, (ICAPS-2007)*. AAAI, (2007).
[21] Sung Wook Yoon, Alan Fern, and Robert Givan, 'Ff-replan: A baseline for probabilistic planning', in *ICAPS*, p. 352. AAAI, (2007).