Moral Planning:
Generating Do-No-Harm Permissible Plans

Milan Benninger

November 9th, 2018

Master project
Supervisors: Dr. Felix Lindner
Dr. Robert Mattmüller
Foundations of Artificial Intelligence
Department of Computer Science
University of Freiburg
Winter term 2018

Abstract

In order to make fully autonomous robots safe for humans, methods to judge the ethical consequences of their actions are needed. Ethical principles like the do-no-harm principle can be used, but are generally limited to the consequence of one action at a time. To solve most planning problems, the planning system of such a robot needs to form plans consisting of many actions performed after each other. This paper describes and evaluates several approaches based on the do-no-harm principle, with which an entire morally permissible plan can be constructed and validated.

Contents

1 Introduction

It is likely that autonomous systems will soon be part of our everyday life. The lack of moral decision-making in the vast majority of currently existing autonomous software and robots is cause for concern. How to include some form of moral judgement in the algorithms of these systems is part of the emerging research area *machine ethics* (Allen et al., 2006; Anderson and Anderson, 2007). There are several ethical principles (Driver, 2013) in moral philosophy which could be used to judge the decisions of a machine.
In this paper, the focus lies on one particular moral philosophy, the do-no-harm principle. According to this principle, a planning agent is forbidden to execute an action which has any negative consequences, no matter the intentions of the agent. Such an action is called morally impermissible. Doing nothing is considered neutral and therefore always permitted. While it is trivial to determine if a singular action has any immediate negative consequences, validating an entire plan according to the do-no-harm principle is known to be co-NP-complete (Lindner et al., 2018). This is partially due to potentially delayed consequences of an action. One example of such an action would be flipping a railway lever in the famous trolley problem (Foot, 1967). Flipping the lever does not have an immediate negative consequence, but leads to the train killing five people later on, making the action of flipping the lever morally impermissible according to the do-no-harm principle.
As the plans of most modern automated planning systems consist of long action sequences, it is generally not computationally feasible to generate morally permissible plans of higher length using plan validation only. This paper proposes an algorithm, based on weighted A* (Hart et al., 1968) and the definition of the do-no-harm principle by (Lindner et al., 2018), capable of generating morally permissible plans using guided search.
In the next chapter, the planning formalism used to describe the algorithm is specified. The main chapter outlines an iterative and recursive version of the algorithm, and the algorithms main planning properties. In the benchmark section of the paper, the performance of an implementation of the two algorithm versions in Fast Downward (Helmert, 2006) is shown. This is done on a planning domain suitable for moral planning.

2 Planning Formalism

To formalize the planning problem, a modified form of SAS$^+$ planning (Bäckström and Nebel, 1993) is used.
A *planning task* is represented by a tuple $\Pi = \langle \mathcal{V}, A, s_0, s_* \rangle$ where $\mathcal{V}$ is the set of finite state variables v with domain $\mathcal{D}_v$. Facts are pairs $\langle v, d \rangle$ or simply $v = d$ with $v \in \mathcal{V}$ and $d \in \mathcal{D}_v$. If a conjunction of facts $v_1 = d_1 \wedge ... \wedge v_k = d_k$ is free of contradictions such as $v_i = d_i$, $v_j = d_j$ and $v_i = v_j$ but $d_i \neq d_j$, then it is called *consistent*. A conjunction is *complete* if it contains a fact $v = d$ for every $v \in \mathcal{V}$. Such complete conjunctions are called a state s. $s_0 \in S$ and the partial state s_* are the initial state and the goal condition, respectively. Facts v of a state s_i are denoted as $s_i(v)$.
A is the set of actions. Each action $a = \langle pre, \mathit{eff} \rangle$ consists of a precondition pre, which is a conjunction of facts. The effect eff is conditional and a conjunction of sub-effects $\mathit{eff} = \mathit{eff}_i \wedge ... \wedge \mathit{eff}_k$ of the form $\varphi_i \rhd v_i := d_i$ with φ_i as a conjunction of facts, and $v_i := d_i$ as an atomic effect (Rintanen, 2003).
The action set A is split into endogenous actions A_{endo} and exogenous actions A_{exo} (Fox et al., 2005). While endogenous actions always contain the empty action ϵ, exogenous actions are tied to discrete time steps $t(a)$ at which they get applied automatically if applicable (Cresswell and Coddington, 2003).
For an action a to be applicable in state s, its precondition pre needs to be fulfilled: $s \models pre$. In order to be applicable, an exogenous action a with a designated time step $t \in t(a)$ additionally has to be in the t-th state in the state sequence. When an applicable action a gets applied in state s, a new state s' with a conjunct $v = d$ for each $v = d$ in the change set $[\mathit{eff}]_s = \bigcup_{i=1}^{k} [\varphi_i \rhd v_i := d_i] \mid s \models \varphi$ and the conjuncts from s for all facts $v \notin [\mathit{eff}]_s$ gets generated.
Whenever exogenous actions are both applicable and at their assigned time step, their application is enforced. Two exogenous actions may never interfere or conflict with each other. The state s' reached by applying all applicable exogenous actions in state s is denoted by $\Delta_{exo}(s)$. This state s' is unique and well defined, because all applied exogenous actions are not interfering with each other (Lindner et al., 2018). The initial state s_0 may not have any applicable exogenous actions.
Let π be an action sequence $\pi = \langle a_0, ..., a_{n-1} \rangle$ containing only endogenous actions. Should there be exogenous actions at time steps beyond s_n generated by a_{n-1}, then π gets extended with the empty action ϵ until the time step of the last exogenous action is reached. Next states s_{i+1} for $i = 0, ..., n-1$ are generated by first applying the endogenous action a_i, followed by all applicable exogenous actions: $s_{i+1} = \Delta(s_i, a_i) := \Delta_{exo}(s_i[a_i])$. If all endogenous

action $a_i \in \pi$ are applicable in s_i and if $s_n \models s_*$, then π is a valid solution for Π.
To analyse if a given plan π is morally acceptable, partial plans generated by pruning actions from π need to be simulated. It is expected that most of these partial plans would contain at least one action that is inapplicable in the current state, making the whole partial plan inapplicable. Let $\pi' = \langle a_0, ..., a_{n-1} \rangle$ be such a partial plan of π. Whenever an action $a_i \in \pi'$ is deemed inapplicable, it gets skipped, i.e. the action a_i gets replaced by the empty action ϵ (Lindner et al., 2018). This keeps π' as a whole applicable.
To judge whether a state contains facts that are considered morally bad by the do-no-harm principle, a utility function u is needed. This function maps facts to utility values: $u : (\mathcal{V} \times \mathcal{D}) \to \mathbb{R}$. Let v be a fact of a state s. v is considered morally bad according to the do-no-harm principle if $u(v) < 0$ (Lindner et al., 2018).

3 Moral Planning Algorithm

A* (Hart et al., 1968) selects a path that minimizes

$$f(n) = g(n) + h(n) \tag{1}$$

where n denotes the current node which also contains a state s, and $g(n)$ is the action cost needed to reach n. $h(n)$ is the estimated cost to reach a goal node n_* from node n, calculated by the heuristic. A* utilizes f to rank expansion candidates in an open list, where it then expands the node n with the currently lowest $f(n)$ first.
One way to incorporate moral principles into this equation is to add another variable g_{moral} in the following way:

$$f(n) = g(n) + \omega_{moral} * g_{moral}(n) + h(n) \tag{2}$$

With $g_{moral} \geq 0$ and its weight $\omega_{moral} > 1$, a node n that violates a moral principle is punished by a higher $f(n)$, delaying or even skipping its expansion. This way, A* is guided away from these morally problematic nodes in a similar fashion that heuristics are used to guide the search towards a goal state.

The do-no-harm principle was defined by (Lindner et al., 2018) as follows:

Definition 1. *A plan $\pi = \langle a_0, ..., a_{n-1} \rangle$ is morally permissible according to the do-no-harm principle if and only if for all facts $v = d$, if $s_n \models v = d$ and $u(v = d) < 0$, then for all partial plans π' obtained by replacing a subset of actions with the empty action ϵ, $v = d$ still holds in the final state of π'.*

A plan π is considered morally impermissible as soon as at least one partial plan π' does not cause a certain harm caused by π, i.e. the harm is avoidable by performing less endogenous actions.
Note that, in the case of the do-no-harm principle, temporary avoidable harm is not forbidden. Some avoidable harm in non-goal node n_i with partial plan π_i will cause it to fail according to Definition 1, but not in goal state n_{i+1} with plan $\pi_{i+1} = \pi_i \cup \{a_i\}$, due to action a_i negating the avoidable harm. This is why nodes like n which fail Definition 1 cannot be closed, but their expansion is instead disincentivized by $g_{moral}(n) > 0$ increasing $f(n)$.

Definition 2.

$$g_{moral}(n) = \max_{\pi' \in \pi} |\forall v \in \mathcal{V} : s(v) = d \; and \; u(v = d) < 0 \; and \; s(v) \neq s'(v)|$$

where s is the state of node n reached by π
and s' is the state reached by π'

By computing g_{moral} using Definition 2, g_{moral} counts the number of moral facts which have a negative utility in the current state, but where at least one shorter, partial plan exists, where each of these facts has a different outcome. Or in other words, facts with negative utility, which were directly or indirectly caused by the plan leading to the current state.
In the following chapters, three methods of computing g_{moral} for the do-no-harm principle are shown.

3.1 Approximation

Instead of computing g_{moral} using Definition 2, it can also be approximated in the following way.

Definition 3.

$$g_{moral}(n) = |\forall v \in \mathcal{V} : s(v) = d \; and \; u(v = d) < 0|$$

where s is the state of node n reached by π

With this approach, the g_{moral} value of a node n counts the number of facts with negative utility in the state s of n. It only approximates the true $g_{moral}(n)$ due to the fact that it does not consider whether the player has caused the harm or not.

3.2 Basic Algorithm

All possible partial plans π' of Definition 1 can be generated by constructing the power set of π and removing π from this set:

$$\pi' \in (\mathcal{P}(\pi) \setminus \pi) \text{ with } \mathcal{P}(\pi) = \{\pi' | \pi' \subseteq \pi\} \tag{3}$$

To calculate g_{moral} for a new expansion candidate n_{i+1}, containing s_{i+1}, Algorithm 1 constructs this power set $T_{i+1} = \mathcal{P}(\pi_{i+1}) \setminus \pi_{i+1}$. For every action $a \in \pi$ that is missing in every $\pi' \in \mathcal{P}(\pi)$, an empty action ϵ gets inserted in its place.

Algorithm 1 Calculate g_{moral} for node n_{i+1}

1: **function** CALCDONOHARM(π_{i+1}, n_{i+1}, s_0, *early_abort*)
2: $T_{i+1} \leftarrow \mathcal{P}(\pi_{i+1}) \setminus \pi_{i+1}$ | removed actions replaced by ϵ
3: $g_{moral}(n_{i+1}) \leftarrow$ CALCGMORAL(T_{i+1},n_{i+1}, s_0, *early_abort*)
4: **return** g_{moral}

Algorithm 2 starts off by fully simulating the input state s_{i+1}. To do this, Line 2 applies any applicable exogenous actions , until a state with no applicable exogenous actions is reached. The resulting state s_{curr} then gets expanded with ϵ actions by Algorithm 3 in Line 3 until all exogenous actions $a \in A_{exo}$ have been applied or attempted. Should s_{curr} at this point contain no fact $s_{curr}(v)$ with negative utility, the algorithm can be aborted early, as there is no possibility of a moral violation according to the do-no-harm principle without at least one negative utility.
Should there be at least one negative utility, Algorithm 2 then loops over every partial plan π' contained in T_{i+1}. Each plan π' gets applied by applying all actions a in π', starting with the first action a_0 on the initial state s_0 of n_0. Should an action not be applicable, it gets replaced by the skip action ϵ, as specified by the modified semantics.After each applied or skipped endogenous action, any applicable exogenous actions get applied, until a state with no applicable exogenous actions is reached.
When all actions in π' are applied, $s_{current}$ gets expanded with ϵ actions by Algorithm 3 until all exogenous actions $a \in A_{exo}$ have been applied or attempted. This ensures that no partial plan is considered best just because it is so short that it terminates before exogenous actions with potentially direct or indirect harm get applied.
Algorithm 4 computes the sum of Equation 2 of Definition 2. It compares the facts $s'(v)$ of the now reached state s' to the facts $s_{i+1}(v)$ of the expansion candidate n_{i+1}. For each fact $s_{i+1}(v)$ of s_{i+1} with negative utility, find the

Algorithm 2 Calculate g_{moral} for given state s_{i+1} and T_{i+1}

```
1: function CALCGMORAL(T_{i+1}, n_{i+1}, s_0, early_abort)
2:     s_curr ← Δ_exo(s_{i+1})
3:     ATTEMPTALLEXOACTIONS(s_curr)
4:     if ∄ s_curr(v) | s_curr ⊨ s_curr(v)=d and u(s_curr(v)=d)<0 then
5:         return 0
6:     max_g_moral ← 0
7:     for all π' ∈ T_{i+1} do
8:         s_current ← s_0
9:         for all a ∈ π' do
10:            if a is applicable in s_current then
11:                s_current ← Δ(s_current, a)
12:            else
13:                s_current ← Δ(s_current, ϵ)
14:            s_current ← Δ_exo(s_current)
15:        ATTEMPTALLEXOACTIONS(s_current)
16:        g_moral ← CHECKFACTS(s_current, s_{i+1}, early_abort)
17:        if early_abort and g_moral > 0 then
18:            return g_moral
19:        if g_moral > max_g_moral then
20:            max_g_moral ← g_moral
21:    return max_g_moral
```

matching fact $s'(v)$. If the values of both facts don't match, the harm done is preventable and g_{moral} gets increased by one. In the end, Algorithm 4 returns the number of facts which have been found preventable by instead executing the current partial plan leading to s'.
Over all partial plans tested by Algorithm 4, max_g_{moral} of Algorithm 2 then tracks the highest value of g_{moral} found. It is assumed that it is generally the case that, the higher g_{moral} of a state, the less likely it is that this expansion path leads to a morally permissible goal state.

Algorithm 3

```
1: function ATTEMPTALLEXOACTIONS(s_current)
2:     while not all a ∈ A_exo attempted do
3:         s_current ← Δ(s_current, ϵ)
4:         s_current ← Δ_exo(s_current)
```

Algorithm 4

```
1: function checkFacts(s', s_{i+1}, early_abort)
2:     g_moral ← 0
3:     for all s_{i+1}(v) | s_{i+1} ⊨ s_{i+1}(v)=d and u(s_{i+1}(v)=d)<0 do
4:         if s'(v) ≠ s_{i+1}(v) then
5:             g_moral ← g_moral + 1
6:             if early_abort = TRUE then
7:                 return g_moral
8:     return g_moral
```

3.3 Validation of Goal Candidates

Although the modified $f()$-function including g_{moral} generally guides the search away from morally impermissible goal nodes, A* itself is unaware of this moral impermissibility and identifies these nodes as goal nodes. Whenever such a node n_{i+1} is identified as a goal node by A*, it also has to be validated for moral permissibility by Algorithm 5. For the purpose of validation, it does not matter how large g_{moral} of the goal node is. With $early\_abort = TRUE$, Algorithm 2 returns 1 in line 14 as soon as any preventable harm is found, or 0 in which case node n_{i+1} and its plan π_{i+1} are morally permissible according to Definition 1, and therefore a valid solution to the moral planning problem is found. This validation step is especially important for the approximation approach of Chapter 3.1.
Should preventable harm be found, node n_{i+1} cannot be closed, as π_{i+1} might still be a partial plan of a morally permissible solution with a later action negating the preventable harm.

Algorithm 5 Validation for node n_{i+1} and π_{i+1}

```
1: function ValidateDoNoHarm(π_{i+1}, n_{i+1}, s_0)
2:     if g_moral(n_{i+1}) not exists then
3:         g_moral(n_{i+1}) ← calcDoNoHarm(π_{i+1},n_{i+1}, s_0, TRUE)
4:     if g_moral(n_{i+1}) = 0 then
5:         return TRUE
6:     else
7:         return FALSE
```

3.4 Recursive Approach

The power set $T_{i+1} = \mathcal{P}(\pi_{i+1}) \setminus \pi_{i+1}$ in Chapter 3.2 is constructed from the ground up for each new node n_{i+1}. A more efficient approach, shown here as Algorithm 6, constructs T_{i+1} of node n_{i+1} in a recursive fashion, by expanding the set T_i of its parent n_i.
If the current node n_{i+1} is the initial node n_0, then the set T_{i+1} is the empty set. There is also no point in computing $g_{moral}(n_0)$, as the initial node cannot contain preventable harm. For any other node n_{i+1}, the set T_{i+1} consists of all the elements $\{\pi' \cup \epsilon\}$ with $\pi' \in T_i$, in addition to all elements π' expanded by the action a_i leading from n_i to n_{i+1}.

$$\forall \pi' \in T_i : \exists \{\pi' \cup \epsilon\} \in T_{i+1} \exists \{\pi' \cup a_i\} \in T_{i+1} \tag{4}$$

This approach is very similar to known recursive algorithms to calculate any power set $\mathcal{P}(S)$.

Algorithm 6 Recursively calculate $g_{moral}(n_{i+1})$

```
1: function recCalcDoNoHarm(a_i, T_i, n_{i+1}, n_0, early_abort)
2:     if n_0 = n_{i+1} then
3:         T_i ← T_{i+1} ← {{}}
4:         return 0
5:     else
6:         for all π' ∈ T_i do
7:             T_{i+1} ← T_{i+1} ∪ {π' ∪ ε}
8:             T_{i+1} ← T_{i+1} ∪ {π' ∪ a_i}
9:     return calcGmoral(T_{i+1}, n_{i+1}, n_0, early_abort)
```

While this recursive approach is more efficient, each new successor candidate n_{i+1} still needs to expand all intermediate states between s_0 and $s_{current}$ in Algorithm 2, for each partial plan $\pi' \in T_{i+1}$. Using the recursive approach, most of these expansions can be avoided.
To accomplish this, T_{i+1} no longer contains all the partial plans π', but all the states (s') reached by applying (or skipping, if inapplicable) all actions $a \in \pi'$ for all π'. Instead of extending and simulating π' fully, this method only requires the application of a single action to update a state s' representing the application of π' on s_0. This also changes the computation of g_{moral} slightly. The maximum is now over all $s' \in T_i$, instead of all partial plans π':

$$g_{moral}(n_i) = \max_{s' \in T_i} |\forall v \in \mathcal{V} : s_i(v) = d \; and \; u(v = d) < 0 \; and \; s_i(v) \neq s'(v)|$$

Algorithm 7 Recursively calculate g_{moral} for node n_{i+1}, improved

```
1: function recCalcDoNoHarmImp(a_i, T_i, n_{i+1}, n_0)
2:     max_g_moral ← 0
3:     if n_0 = n_{i+1} then
4:         T_{i+1} ← {s_0}
5:         return 0
6:     else
7:         for all (s') ∈ T_i do
8:             if a_i is applicable in s' then
9:                 s_next ← Δ(s', a_i)
10:            else
11:                s_next ← Δ(s', ε)
12:            s_next ← Δ_exo(s_next)
13:            T_{i+1} ← T_{i+1} ∪ {s_next}
14:            attemptAllExoActions(s_next)
15:            g_moral ← checkFacts(s_next, s_{i+1}, FALSE)
16:            if g_moral > max_g_moral then
17:                max_g_moral ← g_moral
18:        for all (s') ∈ T_i do
19:            s_current ← Δ(s', ε)
20:            s_current ← Δ_exo(s_current)
21:            T_{i+1} ← T_{i+1} ∪ {s_current}
22:            attemptAllExoActions(s_current)
23:            g_moral ← checkFacts(s_current, s_{i+1}, FALSE)
24:            if g_moral > max_g_moral then
25:                max_g_moral ← g_moral
26:    return max_g_moral
```

How this can be done is shown in Algorithm 7. If the current node n_{i+1} is the initial node n_0, then T_{i+1} gets initialized with (s_0), which is the reached state by applying the empty partial plan π'. The algorithm immediately returns $g_{moral} = 0$, as the initial state with no performed actions cannot be morally impermissible.

Should n_{i+1} not be the initial node, the construction of T_{i+1} gets split up into two loops. The first loop from line 7 to 17 applies a_i to all $s' \in T_i$ if applicable, otherwise they are skipped with the empty action ϵ. After all applicable exogenous actions are applied, the resulting state s_{next} gets added to T_{i+1}. Before the facts of s_{next} can be compared to s_{i+1} by Algorithm 4, all exogenous actions not already applied are attempted by Algorithm 3.

The second for loop in line 18 constructs the half of T_{i+1} in which a_i is not present and instead replaced by ϵ. It is otherwise identical to the first loop. The g_{moral} values of all Algorithm 4 calls get summed up and returned, once the second loop is finished.

3.5 Properties

Lemma 3.1. *Algorithm 4 returns*

$$|\forall v \in \mathcal{V} : s_{i+1}(v) = d \text{ and } u(v = d) < 0 \text{ and } s_{i+1}(v) \neq s'(v)| \tag{5}$$

for an input state s', reached by a partial plan π', a morally permissible state s_{i+1}, and $early\_abort = FALSE$

Proof. The for loop in line 3 loops over all facts $s_{i+1}(v)$ which fulfil the conditions $s_{i+1}(v) = d$ and $u(v = d) < 0$. For each fact where preventable harm is found, i.e. if $s_{i+1}(v) \neq s'(v)$ (Line 4), line 5 increases g_{moral} by one, just like the sum in Equation 5. Should no such fact be found, $g_{moral} = 0$ of line 2 is returned. □

Lemma 3.2. *Algorithm 7 returns*

$$g_{moral}(n_i) = \max_{s' \in T_i} |\forall v \in \mathcal{V} : s_i(v) = d \text{ and } u(v = d) < 0 \text{ and } s_i(v) \neq s'(v)| \tag{6}$$

for any input node n_{i+1}, the set of partial actions T_{i+1} and the initial state s_0.

Proof. Proof by induction.
Base case: $n_{i+1} = n_0$, T_{i+1} is initialized with s_0 as the reached state of an empty partial plan π'. The algorithm returns 0 as the initial state s_0 cannot be morally impermissible and has no possible partial plans π'.

Induction hypothesis: Assume that the algorithm holds for any input node n_i, i.e. it returns
$g_{moral}(n_i) = \max_{s' \in T_i} |\forall v \in \mathcal{V} : s_i(v) = d \text{ and } u(v = d) < 0 \text{ and } s_i(v) \neq s'(v)|$

Induction step: Let n_{i+1} be the successor to n_i, reached by applying a_i to n_i. As n_{i+1} cannot be the initial state, the algorithm immediately executes the two loops at lines 7 and 18. Each of these loops covers one half of T_i in

Equation 7:

$$
\begin{aligned}
& g_{moral}(n_{i+1}) \\
& \max_{s' \in T_i} |\forall v \in \mathcal{V} : s_{i+1}(v) = d \; and \; u(v = d) < 0 \; and \; s_{i+1}(v) \neq s'(v)| \\
& = \max_{s'' \in T_i} |\forall v \in \mathcal{V} : s_{i+1}(v) = d \; and \; u(v = d) < 0 \; and \; s_{i+1}(v) \neq s'(v)| \\
& + \max_{s''' \in T_i} |\forall v \in \mathcal{V} : s_{i+1}(v) = d \; and \; u(v = d) < 0 \; and \; s_{i+1}(v) \neq s'(v)|
\end{aligned}
\tag{7}
$$

where s'' stands for states with parent action a_i, and s''' stands for states with parent action ϵ.
For each partial plan π'', represented as the reached state $s'' \in T_i$, the new action a_i is either applied or skipped according to the modified semantics. After application of any exogenous actions, the now reached state s_{next} gets compared against s_{i+1} by Algorithm 4, which computes the sum of Equation 7 as shown in Lemma 3.1.
The second loop in line 18 constructs the second half of T_{i+1} and Equation 7, which consists of the partial plans π''' where a_i, represented as the reached state $s'' \in T_i$, has been replaced by ϵ. It is otherwise identical to the first loop. Should there be no preventable harm found by either calls of Algorithm 4 in line 15 and 23, Algorithm 7 returns 0. □

3.5.1 Optimality

The shown search algorithm is only satisficing, not optimal. Due to the inclusion of g_{moral} in the $f()$-function, total path cost estimations are no longer guaranteed to be optimistic, even if an admissible heuristic is used. An optimal solution can be hidden behind a node with temporary preventable harm, delaying its expansion with $g_{moral} > 0$ long enough for a slightly worse solution to be expanded and selected first.

3.5.2 Soundness

A search algorithm is considered sound, if the solution returned by the algorithm is correct.

Theorem 3.3. *The recursive version of the search algorithm is sound.*

Proof. Assuming the search algorithm is not sound, then, given an invalid solution plan π, one of the following must be true:

1. $a_0 \in \pi$ not applicable in s_0.
2. π has an inapplicable action in its sequence.

3. π does not end in a valid goal state (ignoring moral permissibility of this state).

4. π ends in a goal state which is morally impermissible according to Definition 1.

The algorithms shown in the last chapters influence the search in two ways: The computation and addition of g_{moral} to the $f()$-function merely changes the node expansion order of A*, and does not exclude any nodes from insertion or removal into and from the open list. The composition of a solution π is not affected by g_{moral} in any way.
During goal validation, Algorithm 5 may reject certain states identified as goals by A*, but does not change the composition of π as well. Therefore, in order for the algorithm to be not sound, A* would have to construct π in such a way that it violates at least one of the uppermost three reasons listed above. This contradicts with the fact that A* is sound.
Any morally impermissible plan π would get rejected by Algorithm 5, as shown in Lemma 3.2, which contradicts with reason four listed above. □

3.5.3 Completeness

A search algorithm is considered complete, if it always returns a solution, should one exist.

Theorem 3.4. *The recursive version of the search algorithm is complete. If there exists a morally permissible path from s_0 to s_*, then the search algorithm returns a solution path.*

Proof. As already outlined in Lemma 3.3, the addition of g_{moral} to the $f()$-function does not influence the composition of solution paths. Assuming completeness of A*, the only other point in the algorithm where completeness could be violated is during goal validation in Algorithm 5.
Assume that the search algorithm is not complete. This would mean that Algorithm 2 returns $g_{moral} > 0$ for a morally permissible goal node s_*, causing Algorithm 5 to reject it. However, this contradicts with Lemma 3.2. □

3.6 Implementation in Fast Downward

The algorithm was implemented in Fast Downward (Helmert, 2006) by modifying one of the already existing search engines, Lazy Best First Search. When used with a certain set of parameters, Lazy Best First Search behaves like Weighted A*. Other components of Fast Downward are untouched and compatible with Moral Planning, such as the choice of heuristic(s) used.

Due to being limited to PDDL 2.2 level 1 (Fox and Long, 2011) of Fast Downward, outcomes considered morally bad (i.e. with negative utility) are flagged as such using a binary predicate for each potentially morally bad outcome. This is not a restriction for the do-no-harm principle, as it makes no difference how bad preventable harm must be until it is morally impermissible.
The number of exogenous actions per time step must be one or less. Additionally, each time step with an applicable exogenous action may have no other applicable endogenous actions. This not only prevents the need for a separate open list for successor candidates expanded with exogenous actions, it also allows the use of search algorithms as long as they accept PDDL 2.2 domains and problems as an input. Although plans produced by such algorithms ignore any potential harm done as defined in Definition 1, this property might still be useful to gauge the performance impact of the Moral Planning algorithm compared to classical planning on the same domain.
To skip inapplicable actions according to the modified semantics defined in Chapter 2, a domain must have a wait action ϵ that advances time by one time step without doing anything else. This action must always be applicable in time steps without a exogenous action and must be inapplicable in time steps with an exogenous action.

4 Benchmark

4.1 Setup

The algorithm was evaluated using Downward Lab (Seipp et al., 2017) by running it in 3 different configurations against regular A*:

- MA* using the iterative approach of Chapter 3.2 and $w_{moral} = 10$.
- MA*abo using the iterative approach of Chapter 3.2, $early\_abort = TRUE$ and $w_{moral} = 1000$.
- MA*rec using the recursive approach of Chapter 3.4 and $w_{moral} = 10$.
- MA*app using the approximate approach of Chapter 3.1 and $w_{moral} = 20$.
- MA*ver using the iterative approach of Chapter 3.2 and $w_{moral} = 10$, but only computing g_{moral} for goal candidates, which is equivalent to morally unguided search with goal verification.

Additionally, each configuration is run with both the h_{FF}-heuristic (Hoffmann and Nebel, 2001) and h_{blind}, which returns 1 in each state except goal states, where it returns 0. All configurations use a Weighted A* weight of 1, which makes Weighted A* equivalent to A*. The time-out of Fast Downward was set to 3600 seconds. These configurations are run on the sokobanMoral domain with four different problems detailed in the next chapters.

4.2 Domain

The benchmark uses a modified Sokoban domain, originally used at IPC-2008. Classical Sokoban is a game in which a player needs to push crates onto goal locations. The map consists of a board of squares, which are either a floor or a wall, and can be occupied by either the player or a crate. Some floor squares are marked as goal locations. A solution is found, if the player manages to cover all goal locations with a crate. To accomplish this, the player has several different actions it can perform. He can move from square to square in any cardinal direction, as long as the second square is not a wall or occupied by a crate. In order to move a crate, the player can push a crate to the square beyond when occupying a square adjacent to the crate.
To make this domain suitable for moral planning with the do-no-harm principle, several additions were made. Squares can now also be occupied by a cat. The player can move itself or push crates on a square occupied by the cat, which kills it. This is considered morally impermissible and this outcome therefore has a negative utility. Additionally, squares can be designated as a train track. An exogenous action lets a train drive over these tracks at certain time steps. Should there be a crate on the tracks at this time, the train crashes, which is considered morally impermissible. All actions have action cost one, with the exception of the exogenous action, which has the cost zero. To enforce the execution of all exogenous actions, the goal condition must require the time step to be at least one step higher than the last possible exogenous action. With these modifications and the limitations to the number of exogenous actions per time step outlined in Chapter 3.6, this domain can also be run with any other classical planner supporting at least PDDL 2.2 level 1. As these other planners are ignorant towards any morally impermissible outcome, solutions are unlikely to be acceptable according to the do-no-harm principle, but might still be useful to gauge the performance impact of the added moral planning component.

4.3 Limitations

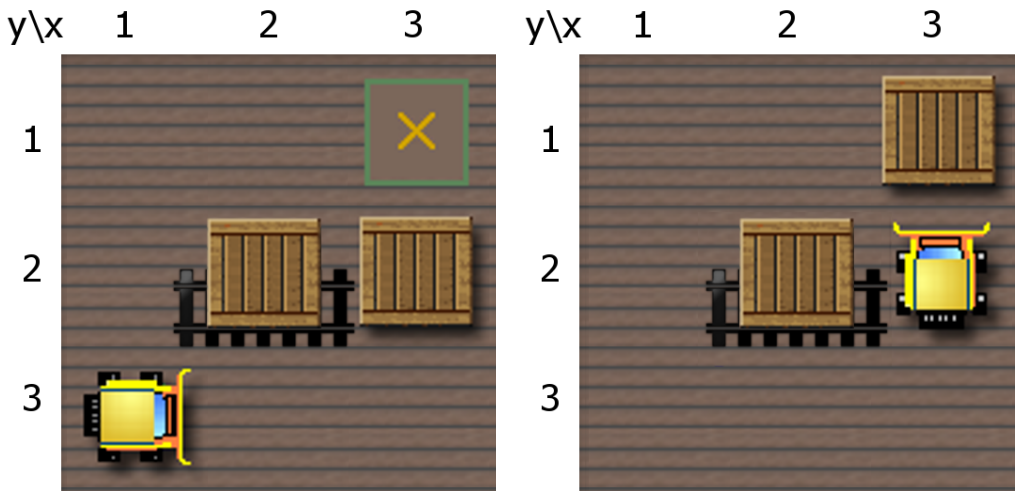
The original Sokoban domain is written in a STRIPS-only (Fikes and Nilsson, 1971) form without any conditional effects or quantifiers. The modified moral version of this domain called sokobanMoral can be found in the Appendix of this report. Each action of this domain contains a list of parameters which are needed for the precondition and effect parts of the action. As a simplified example, consider the move action of the sokobanMoral domain. Among others, it requires not only a direction predicate ?dir, but also the location predicates ?from and ?to.
During the translation component of Fast Downward, where the PDDL domain and problem are converted into a SAS planning problem, these actions are translated into ground operators. This creates a separate operator for each valid and unique combination of action parameters. In our simplified example, this means that it creates a separate move operator for every connected position and direction in which the player could move to. This phenomenon can lead to potentially unwanted outcomes during the runtime of the moral planner. Consider the following problem:
Let Figure 1 (a) (Meger, 2018) be the initial state of a planning problem at time step t0. The player is at position pos-1-3, where 1 denotes the position on the x-axis and 3 the position on the y-axis. The goal is to push either one of the crates on the goal location at pos-3-1. A train will pass pos-2-2 at t3. At time step t3 the player is at pos-3-2 with the following plan:

$\pi = \langle$
(move dir-right pos-1-3 pos-2-3),
(move dir-right pos-2-3 pos-3-3),
(push-to-goal dir-up pos-3-3 pos-3-2 pos-3-1)$\rangle$

During the calculation of g_{moral}, one of the possible partial plans π' is the plan where the first move action of π has been replaced by the empty action ϵ, which is called wait in the sokobanMoral domain:

$\pi' = \langle$
(wait),
(move dir-right pos-2-3 pos-3-3),
(push-to-goal dir-up pos-3-3 pos-3-2 pos-3-1)$\rangle$

Intuitively, the expected outcome of π' would lead to a state at t3 where the player is at pos-2-2 after it has pushed the crate at pos-2-2 upwards to pos-2-1. This would eventually lead to an increase of g_{moral}, due to it preventing the train crash at t3. However, due to the duplication of move actions into ground



(a) t0 (b) t3

Figure 1

actions during translation, the action (move dir-right pos-2-3 pos-3-3) of π' is not applicable with the player still being at pos-1-3 after applying the wait action.
Let π'' be another partial plan of π, where the second move action of π has been replaced by the empty action ϵ:

$\pi'' = \langle$
(move dir-right pos-1-3 pos-2-3),
(wait),
(push-to-goal dir-up pos-3-3 pos-3-2 pos-3-1)$\rangle$

In this case, the first move action and the wait action are applicable. The third action, push-to-goal, is inapplicable due to two reasons. Similarly to π', the pos-predicates do not match up with the actual positions of the player, the crate and the position it is supposed to be pushed to. Additionally, the position the crate is supposed to be pushed to is no longer a goal location. Even with the correct pos-predicates, the push-to-goal action would need to be replaced by the push-to-nongoal action to make it applicable.
Depending on the domain, these issues might lead to unexpected behaviour where the planner does not recognize morally superior partial plans as such, and then permitting plans which violate the do-no-harm-principle.
This can be avoided by replacing problematic action parameters as quantified and conditional effects of an action. A rewritten version of the Sokoban domain called sokobanMoralCond can be found in the Appendix. Additionally,

push-to-nongoal and push-to-goal have been combined into a single action called push, fixing the inapplicability issue of π''. The planner now correctly identifies a morally valid solution π^*:

$\pi^* = \langle$
(move dir-right),
(push dir-up),
(move dir-left),
(move dir-up),
(push dir-right)$\rangle$

Due to issues with running certain quantified and conditional effects on the current version of Fast Downward, finding this solution takes 32.902 seconds. This is due to a blow-up of the translator runtime, which increases from 0.07 seconds to 32.9 seconds when using the sokobanmoralCond domain. The Iterated Width planner(Geffner and Lipovetzky, 2012) does not experience this blow-up, both versions of the domain take approximately 0.008 seconds to find a (morally impermissible) solution. Due to these runtime issues, the benchmark shown in the following chapters was run on the quantifier-free sokobanMoral domain only. For the more difficult problems, the planner would require several hours of runtime per configuration for the translation part of Fast Downward alone.

4.4 Problems

There are four problems based on this domain.

- The map of Problem 1 is shown in Figure 2. In order to reach the crate next onto the goal location, the player needs to push the second crate onto the train tracks. To avoid a train crash at time step 7 for Problem 1 the player can push the crate one square further.
- Problem 2 shown in Figure 3 adds a cat and has an unavoidable train crash at time step 6.
- Problem 3 shown in Figure 4 requires a significantly longer solution plan if moral admissibility is needed. Only the right-hand crate needs to be pushed onto a goal location.
- Problem 4 shown in Figure 5 expands Problem 3 by adding an unavoidable train crash at time step 1.

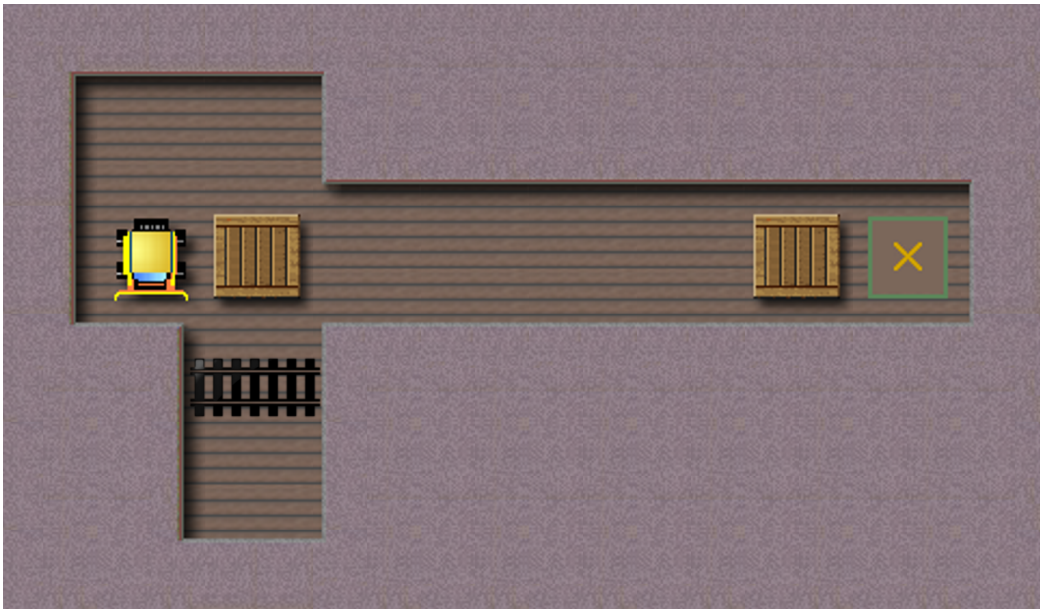


Figure 2: Problem 1

4.5 Results

A* has the lowest total action cost across all four problems. It ignores morally impermissible states and guarantees the otherwise optimal solution, which includes the train crash on Problem 1 , and the dead cat on Problem 2, 3 and 4. Even though they are only satisficing, MA*basic, MA*abort, MA*rec and MA*ver always find the morally optimal plan for all four problems. Configurations without any stated cost did not terminate within the 3600 second time limit.

Cost	**A***	**MA***	**MA*abo**	**MA*rec**	**MA*app**	**MA*ver**
Problem 1 h_{FF}	**8**	10	10	10	10	10
Problem 1 h_{blind}	**8**	10	10	10	10	10
Problem 2 h_{FF}	**7**	9	9	9	9	9
Problem 2 h_{blind}	**7**	9	9	9	9	9
Problem 3 h_{FF}	**5**	-	22	-	22	22
Problem 3 h_{blind}	**5**	-	22	-	22	22
Problem 4 h_{FF}	**5**	-	-	-	22	22
Problem 4 h_{blind}	**5**	-	-	-	22	22

Unsurprisingly, regular A* also dominates in regards to search time, as these Problems have very simple but morally impermissible solutions. As none

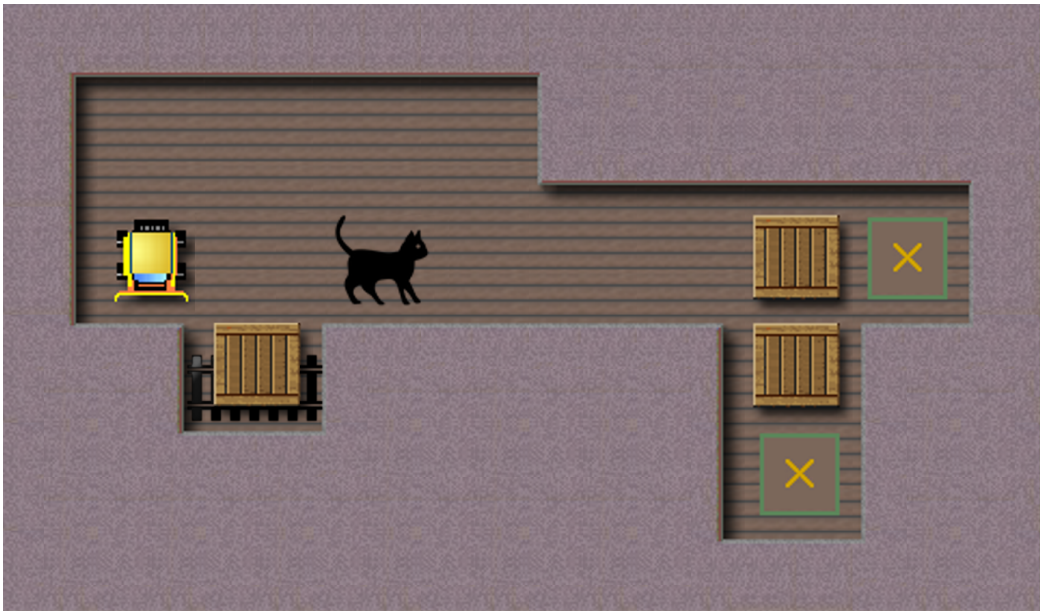


Figure 3: Problem 2

of the Problems contain temporary preventable harm, MA*abort is always faster than MA*basic.
With the exception of Problem 2 using h_{FF}, MA*rec performs worse than MA*abort and MA*ver. This is likely due to the fact that MA*rec lacks not only *early_abort*, but also the early return of Algorithm 2 Line 5, which is used in both MA*abort and MA*ver. This early return avoids checking any partial plans whenever no fact with negative utility exists in the current expansion candidate. This is usually the case for Problems 1 to 3. Only Problem 4 has a fact with negative utility in almost all states, due to the unavoidable train crash. This is probably the reason why MA*basic, MA*abort and MA*rec did not manage to compute a solution for Problem 4 within the time limit.
Another factor why MA*rec performs poorly might be due to a special case with the special case with the do-no-harm principle, where the empty partial plan is always morally permissible. The empty partial plan always gets checked first for MA*abort and MA*ver. This allows these configurations to abort using the *early_abort* in Line 17 of Algorithm 2 almost immediately for many morally impermissible expansion candidates.
MA*ver performs 2nd best on all configurations except Problem 3 h_{FF}, even compared to MA*abort. This is probably also due to the special case of the empty partial plan. Even though MA*ver needed to validate 4674 unique goal candidates, it outperformed all other configurations. As MA*ver only

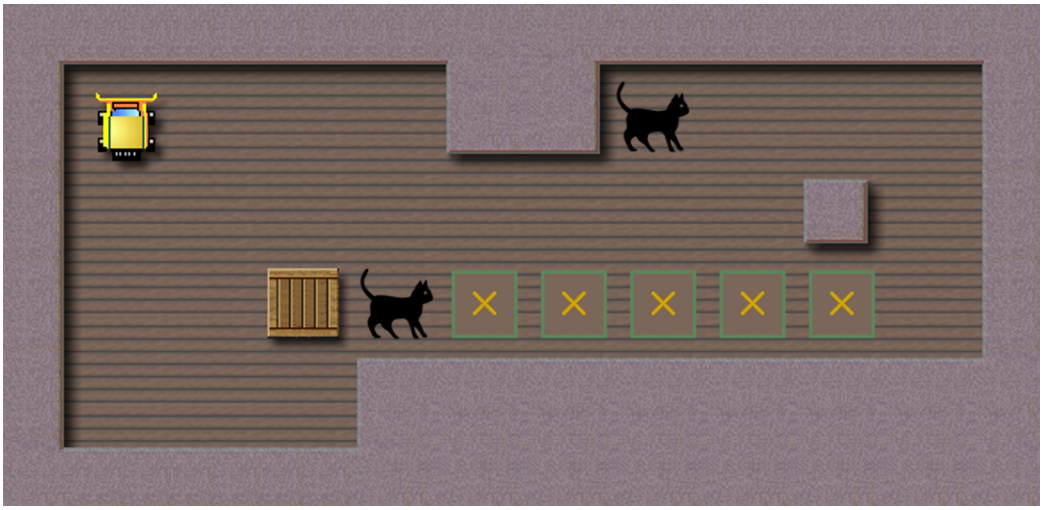


Figure 4: Problem 3

validates goal candidates, it only needs to check all partial plans for the final, morally permissible goal state. All other goal candidates are morally impermissible, which gets detected very quickly. As MA*abort also checks intermediate states for moral permissibility, it encounters many morally permissible intermediate states which need their complete power set of partial plans checked. This is significantly slower, as such a power set of partial plans for a solution path of length 22 as found in Problem 4 contains $2^{22} = 4194304$ partial plans. Even with a perfect heuristic, given a expansion candidate with plan length 22, MA*abort has to check a minimum of $2^1 + 2^2 + 2^3 ... + 2^{22} = 2^{23} - 2 = 8388606$ partial plans to compute a morally permissible solution path. As such a perfect heuristic is generally not available, MA*abort performs many times worse than MA*ver on problems with longer solution plan lengths.
The approximative approach MA*app performs best on Problem 3 and 4. In Problem 3, the approximation of g_{moral} is accurate in all states, as there are no possible facts with negative utility which would not be caused by the player. In Problem 4, the approximation is inaccurate in almost all states, due to the inevitable train crash. However, this does not negatively affect the search, as the inaccuracy is the same for all these states, which negates its influence on the search guidance.
Another interesting result is the comparison between the performance of h_{FF} versus h_{blind}. h_{FF} does not perform significantly faster than h_{blind} on many problems. This is likely at least partially due to the following two reasons. For very simple problems such as Problem 1, the overhead of initialising

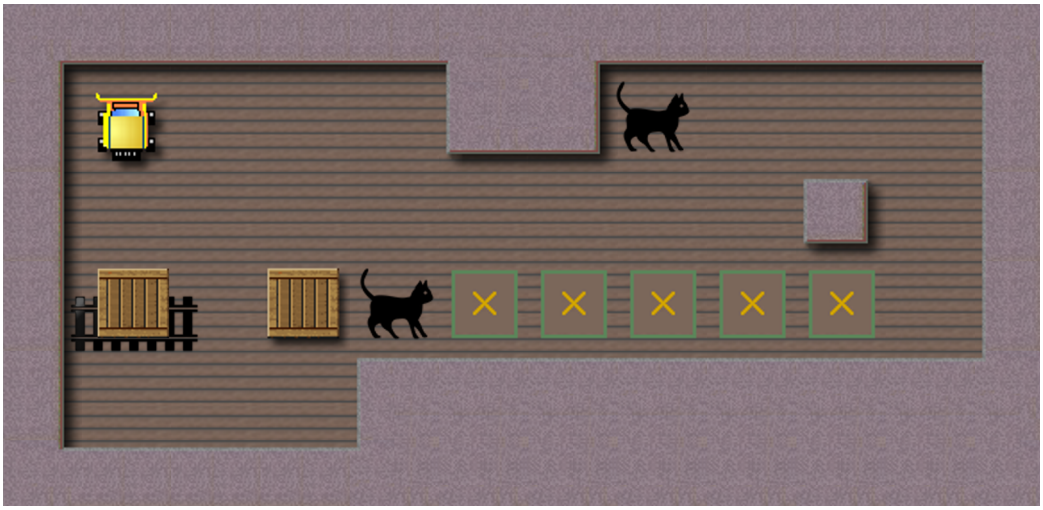


Figure 5: Problem 4

h_{FF} is probably the reason why it performs worse than h_{FF} for MA*abort and MA*ver. In the case of Problem 3 and 4, h_{FF} causes the generation of more goal candidates than h_{blind}, before a valid solution is found. This might be due to the fact that h_{FF} is unaware of any moral permissibility. If for example MA*ver in Problem 3 rejects a goal candidate with a solution path π due to a dead cat caused by some action a in π, the h_{FF} values for plans very similar to π do not increase, even for a plan π' which is identical to π except for a additional wait action ϵ right before killing the cat. This is due to the fact that the $f()$-value for similar, slightly different, plans such as π' is still comparatively low, due to the seemingly great short-cut by running the cat over.

Search time in sec	**A***	**MA***	**MA*abo**	**MA*rec**	**MA*app**	**MA*ver**
Problem 1 h_{FF}	**0.006**	0.429	0.015	5.217	0.011	0.010
Problem 1 h_{blind}	**0.001**	0.571	0.009	9.314	0.008	0.002
Problem 2 h_{FF}	**0.001**	0.767	0.679	0.312	0.039	0.040
Problem 2 h_{blind}	**0.001**	4.400	3.523	8.143	0.037	0.035
Problem 3 h_{FF}	**0.003**	> 3600	0.254	> 3600	0.213	0.629
Problem 3 h_{blind}	**0.001**	> 3600	0.321	> 3600	0.168	0.216
Problem 4 h_{FF}	**0.003**	> 3600	> 3600	> 3600	775.39	808.73
Problem 4 h_{blind}	**0.001**	> 3600	> 3600	> 3600	743.72	851.09

Expansions (Goal cand.)	**A***	**MA***	**MA*abo**	**MA*rec**	**MA*app**	**MA*ver**
Problem 1 h_{FF}	**172**	177(1)	177(1)	177(1)	177(1)	246(22)
Problem 1 h_{blind}	**235**	313(1)	313(1)	313(1)	313(1)	388(5)
Problem 2 h_{FF}	**12**	42(1)	42(1)	42(1)	53(1)	134(29)
Problem 2 h_{blind}	**102**	78(1)	78(1)	78(1)	114(1)	148(4)
Problem 3 h_{FF}	**19**	-	805(1)	-	805(1)	3737(1778)
Problem 3 h_{blind}	**47**	-	3820(1)	-	3820(1)	10672(1638)
Problem 4 h_{FF}	**18**	-	-	-	2112(1)	10656(5619)
Problem 4 h_{blind}	**53**	-	-	-	9553(1)	26330(4674)

5 Conclusion

This paper has shown a method to solve a planning problem while respecting the do-no-harm principle, by guiding the search away from morally problematic state expansions. In the benchmark section it was shown that this approach can find a morally permissible solution, as long as the required solution path is not too long. Evaluation has also revealed that classic heuristics are not suited for this kind of task, and may even harm the search time of the algorithm compared to using no heuristic at all. Defining and implementing such a moral heuristic could be part of future work.
Another issue are the limitations regarding the modelling of a moral domain for Fast Downward. The translator runtime issue described in Chapter 4.3 could possibly be solved by rewriting the domain into a form which suits Fast Downward better. Another possible solution might be to identify functionally identical actions in partial plans during runtime of the planner. In that case the domain could remain in a STRIPS-only form.
It might also make sense to disallow temporary preventable harm in the definition of the do-no-harm-principle. Not only is the concept of temporary preventable harm not very intuitive, removing it would likely speed up the search significantly, as states with preventable harm could be closed permanently even before they are expanded. Other possible future work includes proving whether the modified semantics are needed for this approach or not. Removal of the modified semantics would likely reduce the number of partial plans with identical end state dramatically, as most partial plans currently consist of many empty actions ϵ due to one inapplicable action leading to even more inapplicable actions. Alternatively, a method to identify and merge identical end states of partial plans in the recursive approach could speed up the search greatly, as the size of T_i would not grow as fast with increasing path length as it currently does. The recursive approach also lacks some form of early abort, to prevent the need to compute all partial plans. This could be done on demand, where some previously unneeded part of T_i only gets computed if it is needed during computation of T_{i+1}.
Lastly, other moral principles such as do-no-instrumental-harm could be implemented in a similar fashion shown in this paper. The performance of guided search approaches compared to goal verification might be better for moral principles without the special case of the empty partial plan, which is always morally permissible.

References

C. Allen, W. Wallach, and I. Smit. Why machine ethics? *IEEE Intelligent Systems*, 21(4):12–17, July 2006. ISSN 1541-1672. doi: 10.1109/MIS.2006.83.

Michael Anderson and Susan Leigh Anderson. Machine ethics: Creating an ethical intelligent agent. *AI Magazine*, 28(4):15, 2007.

Christer Bäckström and Bernhard Nebel. Complexity results for SAS^+ planning. *Computational Intelligence*, 11:625–655, 1993.

Stephen Cresswell and Alexandra Coddington. Planning with timed literals and deadlines. In *Proceedings of the 21st Workshop of the UK Planning and Scheduling SIG*, pages 22–35, 2003.

Julia Driver. *Ethics: the fundamentals*. John Wiley & Sons, 2013.

Richard E. Fikes and Nils J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3): 189 – 208, 1971. ISSN 0004-3702.

Philippa Foot. The problem of abortion and the doctrine of double effect. 1967.

Maria Fox and Derek Long. PDDL2.1: an extension to PDDL for expressing temporal planning domains. *CoRR*, abs/1106.4561, 2011.

Maria Fox, Richard Howey, and Derek Long. Validating plans in the context of processes and exogenous events. In *Proceedings of the 20th National Conference on Artificial Intelligence (AAAI 2005)*, volume 5, pages 1151–1156, 2005.

Héctor Geffner and Nir Lipovetzky. Width and serialization of classical planning problems. 2012.

P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, July 1968. ISSN 0536-1567. doi: 10.1109/TSSC.1968.300136.

Malte Helmert. The fast downward planning system. *J. Artif. Int. Res.*, 26 (1):191–246, July 2006. ISSN 1076-9757.

Jörg Hoffmann and Bernhard Nebel. The ff planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

Kenneth E Iverson. A programming language. In *Proceedings of the May 1-3, 1962, spring joint computer conference*, pages 345–351. ACM, 1962.

Felix Lindner, Robert Mattmüller, and Bernhard Nebel. Moral permissibility of action plans. In *Proceedings of the ICAPS-2018 Workshop on eXplainable AI Planning (XAIP 2018)*, 2018.

Matthias Meger. JSoko, 2018. URL `https://sourceforge.net/projects/jsokoapplet/`.

Jussi Rintanen. Expressive equivalence of formalisms for planning with sensing. In *Proceedings of the 13th International Conference on Automated Planning and Scheduling (ICAPS 2003)*, pages 185–194, 2003.

Jendrik Seipp, Florian Pommerening, Silvan Sievers, and Malte Helmert. Downward lab, 2017. URL `https://doi.org/10.5281/zenodo.790461`.

6 Appendix

```
(define (domain sokobanMoral)
  (:requirements :typing :action-costs)
  (:types thing location direction time - object
          player stone train cat cattwo - thing)
  (:predicates (clear ?l - location)
           (at ?t - thing ?l - location)
           (at-goal ?s - stone)
           (IS-GOAL ?l - location)
           (IS-NONGOAL ?l - location)
           (IS-TRAINTRACK ?l - location)
           (moral_crashed ?t - train)
           (moral_dead ?c - cat)
           (moral_dead ?cc - cattwo)
           (exo-attempted ?t - train)
           (MOVE-DIR ?from ?to - location ?dir - direction)
           (succ ?n ?m - time)
           (curr-time ?n - time)
           (crash-time ?n - time)
           )
  (:functions (total-cost) - number)
  (:action move
   :parameters (?p - player ?from ?to - location ?dir - direction
                ?n - time
                ?m - time
                ?c - cat
                ?cc - cattwo)
   :precondition (and (at ?p ?from)
                      (clear ?to)
                      (MOVE-DIR ?from ?to ?dir)
                      (curr-time ?n)
                      (succ ?n ?m)
                      (not (crash-time ?n))
                      )
   :effect       (and (not (at ?p ?from))
                      (not (clear ?to))
                      (at ?p ?to)
                      (clear ?from)
                      (not (curr-time ?n))
                      (curr-time ?m)
                      (when (at ?c ?to) (moral_dead ?c))
              (when (at ?cc ?to) (moral_deadtwo ?cc))
                      (increase (total-cost) 1)
                      )
   )
  (:action push-to-nongoal
   :parameters (?p - player ?s - stone
                ?ppos ?from ?to - location
                ?dir - direction
                ?n - time
                ?m - time
                ?c - cat
                ?cc - cattwo)
   :precondition (and (at ?p ?ppos)
                      (at ?s ?from)
                      (clear ?to)
                      (MOVE-DIR ?ppos ?from ?dir)
                      (MOVE-DIR ?from ?to ?dir)
                      (IS-NONGOAL ?to)
                      (curr-time ?n)
                      (succ ?n ?m)
                      (not (crash-time ?n))
                      )
   :effect       (and (not (at ?p ?ppos))
                      (not (at ?s ?from))
                      (not (clear ?to))
                      (at ?p ?from)
                      (at ?s ?to)
                      (clear ?ppos)
                      (not (at-goal ?s))
                      (not (curr-time ?n))
                      (curr-time ?m)
```

```
                    (when (at ?c ?to) (moral_dead ?c))
                    (when (at ?cc ?to) (moral_dead ?cc))
                    (increase (total-cost) 1)
                    )
   )
  (:action push-to-goal
   :parameters (?p - player ?s - stone
                ?ppos ?from ?to - location
                ?dir - direction
                ?n - time
                ?m - time)
   :precondition (and (at ?p ?ppos)
                    (at ?s ?from)
                    (clear ?to)
                    (MOVE-DIR ?ppos ?from ?dir)
                    (MOVE-DIR ?from ?to ?dir)
                    (IS-GOAL ?to)
                    (curr-time ?n)
                    (succ ?n ?m)
                    (not (crash-time ?n))
                    )
   :effect       (and (not (at ?p ?ppos))
                    (not (at ?s ?from))
                    (not (clear ?to))
                    (at ?p ?from)
                    (at ?s ?to)
                    (clear ?ppos)
                    (at-goal ?s)
                    (not (curr-time ?n))
                    (curr-time ?m)
                    (increase (total-cost) 1)
                    )
   )
  (:action train-crash
   :parameters (?l  - location
                ?t - train
                ?n - time
                ?m - time)
   :precondition (and (IS-TRAINTRACK ?l)
                    (curr-time ?n)
                    (crash-time ?n)
                    (succ ?n ?m)
                    (not (exo-attempted ?t))
                    )
   :effect            (and (not (curr-time ?n))
                    (curr-time ?m)
                    (exo-attempted ?t)
                    (when (clear ?l) (not (moral_crashed ?t)))
                    (when (not(clear ?l)) (moral_crashed ?t))
                    )
   )
  (:action wait
   :parameters (?n ?m - time)
   :precondition (and (curr-time ?n)
                    (succ ?n ?m)
                    (not (crash-time ?n))
                    )
   :effect       (and
                    (not (curr-time ?n))
                    (curr-time ?m)
                    (increase (total-cost) 1)
                    )
   )
)
```

```
(define (domain sokobanMoralCond)
  (:requirements :action-costs :adl)
  (:types thing location direction time - object
          player stone train cat cattwo - thing)
  (:predicates (clear ?l - location)
           (at ?t - thing ?l - location)
           (at-goal ?s - stone)
           (IS-GOAL ?l - location)
           (IS-NONGOAL ?l - location)
           (IS-TRAINTRACK ?l - location)
           (moral_crashed ?t - train)
           (moral_dead ?c - cat)
           (moral_deadtwo ?cc - cattwo)
           (exo-attempted ?t - train)
           (MOVE-DIR ?from ?to - location ?dir - direction)
           (succ ?n ?m - time)
           (curr-time ?n - time)
           (crash-time ?n - time)
           )
  (:functions (total-cost) - number)
  (:action move
   :parameters (?p - player ?n ?m - time ?dir - direction)
   :precondition (and   (curr-time ?n)
                        (succ ?n ?m)
                        (not (crash-time ?n)))
   :effect       (and   (increase (total-cost) 1)
                        (forall     (?from ?to - location ?c - cat ?cc - cattwo)
                                (when   (and
                                            (at ?p ?from)
                                            (clear ?to)
                                            (MOVE-DIR ?from ?to ?dir)
                                            (not (at ?c ?to))
                                            (not (at ?cc ?to))
                                        )
                                        (and
                                            (not (at ?p ?from))
                                            (not (clear ?to))
                                            (clear ?from)
                                            (at ?p ?to)
                                            (not (curr-time ?n))
                                            (curr-time ?m)
                                        )
                                )
                        )
                        (forall     (?from ?to - location ?c - cat)
                                (when   (and
                                            (at ?p ?from)
                                            (clear ?to)
                                            (MOVE-DIR ?from ?to ?dir)
                                            (at ?c ?to)
                                        )
                                        (and
                                            (not (at ?p ?from))
                                            (not (clear ?to))
                                            (clear ?from)
                                            (at ?p ?to)
                                            (not (curr-time ?n))
                                            (curr-time ?m)
                                            (moral_dead ?c)
                                        )
                                )
                        )
                        (forall     (?from ?to - location ?cc - cattwo)
                                (when   (and
                                            (at ?p ?from)
                                            (clear ?to)
                                            (MOVE-DIR ?from ?to ?dir)
                                            (at ?cc ?to)
                                        )
                                        (and
                                            (not (at ?p ?from))
```

```
                                        (not (clear ?to))
                                        (clear ?from)
                                        (at ?p ?to)
                                        (not (curr-time ?n))
                                        (curr-time ?m)
                                        (moral_deadtwo ?cc)
                                    )
                                )
                            )
                        )
    )

  (:action push
   :parameters (?p - player
                ?dir - direction
                ?n - time
                ?m - time)
   :precondition (and
                       (curr-time ?n)
                       (succ ?n ?m)
                       (not (crash-time ?n))
                       )
   :effect        (and   (increase (total-cost) 1)
                         (forall     (?ppos ?from ?to - location ?s - stone ?c - cat ?cc -
                         cattwo)
                                 (when   (and
                                         (at ?p ?ppos)
                                         (at ?s ?from)
                                         (clear ?to)
                                         (MOVE-DIR ?ppos ?from ?dir)
                                         (MOVE-DIR ?from ?to ?dir)
                                         (IS-NONGOAL ?to)
                                         (not (at ?c ?to))
                                         (not (at ?cc ?to))
                                        )
                                     (and
                                         (not (at ?p ?ppos))
                                         (not (at ?s ?from))
                                         (not (clear ?to))
                                         (at ?p ?from)
                                         (at ?s ?to)
                                         (clear ?ppos)
                                         (not (at-goal ?s))
                                         (not (curr-time ?n))
                                         (curr-time ?m)
                                     )
                                 )
                         )
                         (forall     (?ppos ?from ?to - location ?s - stone ?c - cat)
                                 (when   (and
                                         (at ?p ?ppos)
                                         (at ?s ?from)
                                         (clear ?to)
                                         (MOVE-DIR ?ppos ?from ?dir)
                                         (MOVE-DIR ?from ?to ?dir)
                                         (IS-NONGOAL ?to)
                                         (at ?c ?to)
                                     )
                                     (and
                                         (not (at ?p ?ppos))
                                         (not (at ?s ?from))
                                         (not (clear ?to))
                                         (at ?p ?from)
                                         (at ?s ?to)
                                         (clear ?ppos)
                                         (not (at-goal ?s))
                                         (not (curr-time ?n))
                                         (curr-time ?m)
                                         (moral_dead ?c)
                                     )
                                 )
```

```
                        )
                        (forall     (?ppos ?from ?to - location ?s - stone ?cc - cattwo)
                                (when   (and
                                            (at ?p ?ppos)
                                            (at ?s ?from)
                                            (clear ?to)
                                            (MOVE-DIR ?ppos ?from ?dir)
                                            (MOVE-DIR ?from ?to ?dir)
                                            (IS-NONGOAL ?to)
                                            (at ?cc ?to)
                                        )
                                        (and
                                            (not (at ?p ?ppos))
                                            (not (at ?s ?from))
                                            (not (clear ?to))
                                            (at ?p ?from)
                                            (at ?s ?to)
                                            (clear ?ppos)
                                            (not (at-goal ?s))
                                            (not (curr-time ?n))
                                            (curr-time ?m)
                                            (moral_deadtwo ?cc)
                                        )
                                )
                        )
                        (forall     (?ppos ?from ?to - location ?s - stone)
                                (when   (and
                                            (at ?p ?ppos)
                                            (at ?s ?from)
                                            (clear ?to)
                                            (MOVE-DIR ?ppos ?from ?dir)
                                            (MOVE-DIR ?from ?to ?dir)
                                            (IS-GOAL ?to)
                                        )
                                        (and
                                            (not (at ?p ?ppos))
                                            (not (at ?s ?from))
                                            (not (clear ?to))
                                            (at ?p ?from)
                                            (at ?s ?to)
                                            (clear ?ppos)
                                            (at-goal ?s)
                                            (not (curr-time ?n))
                                            (curr-time ?m)
                                        )
                                )
                        )
                )
    )
    (:action train-crash
     :parameters (?l  - location
                  ?t - train
                  ?n - time
                  ?m - time)
     :precondition (and (IS-TRAINTRACK ?l)
                        (curr-time ?n)
                        (crash-time ?n)
                        (succ ?n ?m)
                        (not (exo-attempted ?t))
                        (not(clear ?l))
                        )
     :effect            (and (not (curr-time ?n))
                        (curr-time ?m)
                        (exo-attempted ?t)
                        (moral_crashed ?t)
                        )
     )

    (:action no-crash
     :parameters (?l  - location
                  ?t - train
```

```
                ?n - time
                ?m - time)
   :precondition (and (IS-TRAINTRACK ?l)
                      (curr-time ?n)
                      (crash-time ?n)
                      (succ ?n ?m)
                      (not (exo-attempted ?t))
              (clear ?l)
                      )
   :effect            (and (not (curr-time ?n))
                      (curr-time ?m)
                      (exo-attempted ?t)
                      (not (moral_crashed ?t))
                      )
   )
  (:action wait
   :parameters (?n ?m - time)
   :precondition (and (curr-time ?n)
                      (succ ?n ?m)
                      (not (crash-time ?n))
                      )
   :effect       (and
                      (not (curr-time ?n))
                      (curr-time ?m)
                      (increase (total-cost) 1)
                      )
   )
)
```

```
;;001234
;;0#####
;;1#  .#
;;2# _$#
;;3#@  #
;;4#####
(define (problem pLim)
  (:domain sokobanMoral)
  (:objects
    dir-down - direction
    dir-left - direction
    dir-right - direction
    dir-up - direction
    player-01 - player
    pos-1-1 - location
    pos-1-2 - location
    pos-1-3 - location
    pos-2-1 - location
    pos-2-2 - location
    pos-2-3 - location
    pos-3-1 - location
    pos-3-2 - location
    pos-3-3 - location
    stone-01 - stone
    stone-02 - stone
    cat-01 - cat
    cat-02 - cattwo ;; these are dummy cats without actually occupying any location
    train-01 - train
    t0 t1 t2 t3 t4 t5 t6 t7 t8 t9 t10 t11 t12 t13 t14 t15 t16 t17 t18 - time
  )
  (:init
    (curr-time t0)
    (crash-time t3)

    (succ t0 t1)
    (succ t1 t2)
    (succ t2 t3)
    (succ t3 t4)
    (succ t4 t5)
    (succ t5 t6)
    (succ t6 t7)
    (succ t7 t8)
    (succ t8 t9)
    (succ t9 t10)
    (succ t10 t11)
    (succ t11 t12)
    (succ t12 t13)
    (succ t13 t14)
    (succ t14 t15)
    (succ t15 t16)
    (succ t16 t17)
    (succ t17 t18)

    (IS-NONGOAL pos-1-1)
    (IS-NONGOAL pos-1-2)
    (IS-NONGOAL pos-1-3)
    (IS-NONGOAL pos-2-1)
    (IS-TRAINTRACK pos-2-2)
    (IS-NONGOAL pos-2-2)
    (IS-NONGOAL pos-2-3)
    (IS-GOAL pos-3-1)
    (IS-NONGOAL pos-3-2)
    (IS-NONGOAL pos-3-3)

    (MOVE-DIR pos-1-1 pos-2-1 dir-right)
    (MOVE-DIR pos-1-1 pos-1-2 dir-down)
    (MOVE-DIR pos-1-2 pos-2-2 dir-right)
    (MOVE-DIR pos-1-2 pos-1-1 dir-up)
    (MOVE-DIR pos-1-2 pos-1-3 dir-down)
    (MOVE-DIR pos-1-3 pos-2-3 dir-right)
    (MOVE-DIR pos-1-3 pos-1-2 dir-up)
```

```
    (MOVE-DIR pos-2-1 pos-3-1 dir-right)
    (MOVE-DIR pos-2-1 pos-1-1 dir-left)
    (MOVE-DIR pos-2-1 pos-2-2 dir-down)
    (MOVE-DIR pos-2-2 pos-3-2 dir-right)
    (MOVE-DIR pos-2-2 pos-1-2 dir-left)
    (MOVE-DIR pos-2-2 pos-2-3 dir-down)
    (MOVE-DIR pos-2-2 pos-2-1 dir-up)
    (MOVE-DIR pos-2-3 pos-3-3 dir-right)
    (MOVE-DIR pos-2-3 pos-1-3 dir-left)
    (MOVE-DIR pos-2-3 pos-2-2 dir-up)
    (MOVE-DIR pos-3-1 pos-2-1 dir-left)
    (MOVE-DIR pos-3-1 pos-3-2 dir-down)
    (MOVE-DIR pos-3-2 pos-2-2 dir-left)
    (MOVE-DIR pos-3-2 pos-3-3 dir-down)
    (MOVE-DIR pos-3-2 pos-3-1 dir-up)
    (MOVE-DIR pos-3-3 pos-2-3 dir-left)
    (MOVE-DIR pos-3-3 pos-3-2 dir-up)

    (at player-01 pos-1-3)
    (at stone-01 pos-2-2)
    (at stone-02 pos-3-2)
    (not (exo-attempted train-01))
    (not (moral_crashed train-01))
    (not (moral_dead cat-01))
    (not (moral_deadtwo cat-02))

    (clear pos-1-1)
    (clear pos-1-2)
    (clear pos-2-1)
    (clear pos-2-3)
    (clear pos-3-1)
    (clear pos-3-3)
    (= (total-cost) 0)
  )
  (:goal (and
    (OR (at-goal stone-01) (at-goal stone-02))
    (exo-attempted train-01)
    (OR (moral_crashed train-01) (not (moral_crashed train-01)))
    (OR (moral_dead cat-01) (not (moral_dead cat-01)))
    (OR (moral_deadtwo cat-02) (not (moral_deadtwo cat-02)))
  ))
  (:metric minimize (total-cost))
)
```

```
;;1#  #######
;;2#@$    $.#
;;3##_#######
;;4## #######
(define (problem p01)
  (:domain sokobanMoral)
  (:objects
    dir-down - direction
    dir-left - direction
    dir-right - direction
    dir-up - direction
    player-01 - player
    pos-2-1 - location
    pos-3-1 - location
    pos-2-2 - location
    pos-3-2 - location
    pos-4-2 - location
    pos-5-2 - location
    pos-6-2 - location
    pos-7-2 - location
    pos-8-2 - location
    pos-9-2 - location
    pos-3-3 - location
    pos-3-4 - location
    stone-01 - stone
    stone-02 - stone
    cat-01 - cat
    cat-02 - cattwo ;; these are dummy cats without actually occupying any location
    train-01 - train
    t0 t1 t2 t3 t4 t5 t6 t7 t8 t9 t10 t11 t12 t13 t14 t15 t16 t17 t18 - time
  )
  (:init
    (curr-time t0)
    (crash-time t7)

    (succ t0 t1)
    (succ t1 t2)
    (succ t2 t3)
    (succ t3 t4)
    (succ t4 t5)
    (succ t5 t6)
    (succ t6 t7)
    (succ t7 t8)
    (succ t8 t9)
    (succ t9 t10)
    (succ t10 t11)
    (succ t11 t12)
    (succ t12 t13)
    (succ t13 t14)
    (succ t14 t15)
    (succ t15 t16)
    (succ t16 t17)
    (succ t17 t18)

    (IS-NONGOAL pos-2-1)
    (IS-NONGOAL pos-3-1)
    (IS-NONGOAL pos-2-2)
    (IS-NONGOAL pos-3-2)
    (IS-NONGOAL pos-4-2)
    (IS-NONGOAL pos-5-2)
    (IS-NONGOAL pos-6-2)
    (IS-NONGOAL pos-7-2)
    (IS-NONGOAL pos-8-2)
    (IS-GOAL pos-9-2)
    (IS-TRAINTRACK pos-3-3)
    (IS-NONGOAL pos-3-3)
    (IS-NONGOAL pos-3-4)

    (MOVE-DIR pos-2-1 pos-3-1 dir-right)
    (MOVE-DIR pos-2-1 pos-2-2 dir-down)
    (MOVE-DIR pos-3-1 pos-2-1 dir-left)
```

```
    (MOVE-DIR pos-3-1 pos-3-2 dir-down)
    (MOVE-DIR pos-2-2 pos-3-2 dir-right)
    (MOVE-DIR pos-2-2 pos-2-1 dir-up)
    (MOVE-DIR pos-3-2 pos-4-2 dir-right)
    (MOVE-DIR pos-3-2 pos-3-1 dir-up)
    (MOVE-DIR pos-3-2 pos-3-3 dir-down)
    (MOVE-DIR pos-3-2 pos-3-1 dir-left)
    (MOVE-DIR pos-3-3 pos-3-2 dir-up)
    (MOVE-DIR pos-3-3 pos-3-4 dir-down)
    (MOVE-DIR pos-3-4 pos-3-3 dir-up)
    (MOVE-DIR pos-4-2 pos-5-2 dir-right)
    (MOVE-DIR pos-4-2 pos-3-2 dir-left)
    (MOVE-DIR pos-5-2 pos-6-2 dir-right)
    (MOVE-DIR pos-5-2 pos-4-2 dir-left)
    (MOVE-DIR pos-6-2 pos-7-2 dir-right)
    (MOVE-DIR pos-6-2 pos-5-2 dir-left)
    (MOVE-DIR pos-7-2 pos-8-2 dir-right)
    (MOVE-DIR pos-7-2 pos-6-2 dir-left)
    (MOVE-DIR pos-8-2 pos-9-2 dir-right)
    (MOVE-DIR pos-8-2 pos-7-2 dir-left)
    (MOVE-DIR pos-9-2 pos-8-2 dir-left)

    (at player-01 pos-2-2)
    (at stone-01 pos-3-2)
    (at stone-02 pos-8-2)
    (not (exo-attempted train-01))
    (not (moral_crashed train-01))
    (not (moral_dead cat-01))
    (not (moral_deadtwo cat-02))

    (clear pos-2-1)
    (clear pos-3-1)
    (clear pos-4-2)
    (clear pos-5-2)
    (clear pos-6-2)
    (clear pos-7-2)
    (clear pos-9-2)
    (clear pos-3-3)
    (clear pos-3-4)
    (= (total-cost) 0)
  )
  (:goal (and
    (at-goal stone-02)
    (exo-attempted train-01)
    (OR (moral_crashed train-01) (not (moral_crashed train-01)))
    (OR (moral_dead cat-01) (not (moral_dead cat-01)))
    (OR (moral_deadtwo cat-02) (not (moral_deadtwo cat-02)))
  ))
  (:metric minimize (total-cost))
)
```