Explainable Planning via Counterfactual Task Analysis for the Beluga Challenge and Beyond

Elliot Gestrin[1], **Gustaf Söderholm**[1], **Paul Höft**[1], **Mauricio Salerno**[2], **Jendrik Seipp**[1], **Daniel Gnad**[1]

[1]: Linköping University, Sweden
{firstname}.{lastname}@liu.se

[2]: Universidad Carlos III de Madrid, Spain
msalerno@pa.uc3m.es

Abstract

The Beluga Challenge, recently organized by the Tuples consortium, offered a track on explainable planning (XAIP), to the best of our knowledge the first XAIP competition to date. Within the setting of the Beluga logistics domain, participants were given a planning task and a plan, and were supposed to answer a query to explain to a human expert certain choices made in the plan. The queries ask about particular state atoms that were achieved and alternatives "why achieve *this* atom A instead of *that* atom B?", action reordering "can I do A *before* B instead?", or about the consequences of object removal "what happens if we *forbid* to use object X?". In this work, we propose counterfactual reasoning to come up with explanations that answer these queries. We design task reformulations, modifications that alter the input planning task, such that the solutions for the modified task allow to explain the choices made in the initial plan. Our framework generalizes the queries posed in the Beluga challenge. To obtain textual explanations, we employ a large language model (LLM) that allows our system to be used without planning-specific knowledge. We empirically show that solving the modified task is similarly hard as finding a plan for the original task, showing that our approach is efficient for practical usage.

Introduction

The Beluga AI Challenge (Tuples 2025) is a competition that was hosted by the Tuples[1] consortium with the goal of progressing towards trustworthy AI systems. It does so by bringing AI planning into a real-world scenario, specifically the logistics at Airbus for the construction of planes and use of Beluga airplanes. This is modeled as a classical planning problem where the goal is to construct a set of aircraft from parts delivered by a sequence of Beluga airplanes. We focus on the explainability track of the competition, where participants were asked to explain certain choices made in an input plan for a given task. Since manually solving the logistics tasks is very challenging, one of the main goals of the Beluga Challenge is to explore (semi-)automated approaches that (1) generate plans for Beluga planning tasks and (2) allow interaction by domain experts to explain and possibly modify the solution obtained by an automated planner.

We propose an approach based on counterfactual reasoning to come up with good explanations. While we take inspiration from the concrete queries asked in the competition, we derive a general framework that can be instantiated with these queries, but allows the same *types* of queries in arbitrary domains. Concretely, our framework supports the following three query types: (1) asking about particular state atoms that were achieved and alternatives "why achieve *this* atom A instead of *that* atom B?", (2) action reordering "can I do A *before* B instead?", or (3) about the consequences of object removal "what happens if we *forbid* to use object X?". Many practically relevant questions are covered by these queries, such as "what happens if one of my trucks had an accident and cannot be used?" or "is it possible to put on my right sock *before* the left one instead?". We formalize the task modifications and show how solutions obtained for the modified problem can be used to give explanations that answer the query. For practical usability, we connect the reasoning to a large language model (LLM) that allows our tool to be used by lay persons not familiar with the intricacies of PDDL and classical planning techniques. As an indication that our approach is efficient and usable in practice, we empirically show that solving the modified tasks is usually no harder than solving the original one, and a single additional plan per query suffices to come up with an explanation.

Background

We use a first-order planning formalism, where a planning task is a pair $\Pi = \langle D, I \rangle$, where D is the *domain* and I is the *problem instance* (Corrêa et al. 2020).

A domain is a tuple $D = \langle \mathcal{H}, \mathcal{C}, \mathcal{P}, \mathcal{A} \rangle$, where: $\mathcal{H}$ is the type hierarchy; $\mathcal{C}$ is a set of constants; $\mathcal{P}$ is a set of predicates, where each one has a name and arity, and a type of each argument. $\mathcal{A}$ is a set of action schemas. For an n-ary predicate $p \in \mathcal{P}$ and $t = \langle t_1, \dots, t_n \rangle$, a tuple of typed constants or typed free variables, $p(t)$ is an *atom*, and it is a *ground* atom if t does not contain free variables. An action schema $a \in \mathcal{A}$ is a tuple $a = \langle name(a), params(a), pre^+(a), pre^-(a), add(a), del(a), cost(a) \rangle$, specifying its *name*; the action *parameters*, which is a set of typed free variables; $pre^+(a)$ and $pre^-(a)$ are sets of atoms, defining the positive and negative preconditions for the action, respectively; the *add* and *delete* sets describe what changes after applying the action (added and deleted atoms). $cost(a)$ is a non-negative *cost*.

A problem instance $I = \langle \mathcal{O}, \mathcal{I}, \mathcal{G} \rangle$ is a tuple, where $\mathcal{O}$

[1]https://tuples.ai/

is a set of typed constants, representing the *objects* of the problem; $\mathcal{I}$ and $\mathcal{G}$ are sets of ground atoms describing the *initial state* and the *goals*, respectively.

Ground actions $\overline{a}$ are obtained from action schemas a by instantiating the free variables in the parameters with constants from the problem instance. A ground action is applicable in a state s if $pre^+(\overline{a}) \subseteq s$ and $pre^-(\overline{a}) \cap s = \emptyset$. The successor state s' resulting from applying $\overline{a}$ in s is defined as $s' = (s \setminus del(\overline{a})) \cup add(\overline{a})$. A plan $\pi = \langle a_1, \dots, a_n \rangle$ is a sequence of ground actions such that a_i $(i = 2 \dots n)$ is applicable in the state s_{i-1} generated after applying $a_1, \dots, a_{i-1}$ to $\mathcal{I}$; a_1 is applicable in $\mathcal{I}$; and the consecutive application of all actions in π to $\mathcal{I}$ produces a state s_n such that $G \subseteq s_n$. The cost for a plan is $cost(\pi) = \sum_{a \in \pi} cost(a)$. A plan is *optimal* if no other plan has a lower cost.

The Beluga Logistics Domain

In the Beluga logistics domain, a sequence of Beluga airplanes transport parts of to-be-constructed planes to an assembly facility. All airplane parts are transported on jigs J that come in categories of different sizes $|J|$. Each Beluga delivers a possibly empty sequence of jigs that can only be unloaded in order. The aircraft are assembled in hangars, each requiring parts of specific types in a fixed and known order. Since the arrival order of the aircraft parts may differ from that in which the hangars need them, there exists a storage and reordering system in the form of racks. Each rack R acts as a double-sided queue for jigs up to a capacity limit $|R_i|$, which enforces $\sum_{J \in R} |J| \leq |R|$. The jigs are transported between the Beluga airplanes, racks, and hangars with trailers. Each trailer can transport only one jig of any size at a time and can operate only on one side of the racks: either the side where the Beluga airplanes are unloaded or the other side where the jigs are delivered to the hangars; they are not able to switch sides. Once an aircraft part is delivered to a hangar, the empty jig remains behind and must be managed. Empty jigs need to be transported back to the Beluga airplanes, to be delivered to another site, which is specified by a list of empty jigs that need to be loaded before a Beluga can leave. There can only be one Beluga airplane on site at a time. Figure 1 illustrates a potential scenario. Since the number of trailers on each side and the capacity of each rack is limited and jigs can only be placed and removed from the ends of each rack, while full jigs need to travel to the hangars and empty jigs need to travel to the Belugas, jigs can easily start blocking each other. This makes preventing a lot of shuffling or even dead ends a challenging planning problem. Good solutions minimize the number of *swaps*: operations where jigs are only moved from rack to rack on one side to gain access to a blocked jig. In the real world, flight schedules might also change, so keeping one rack empty as a backup helps to find more robust plans.

The competition features two tracks, one focused on explainability and the other on scalability. We focus on the explainability track. In this track, a planning system has been used to generate a solution for a given task. A human user then asks questions about this plan. The main objective is to answers these queries and thereby explain the initial solution. The possible queries are:

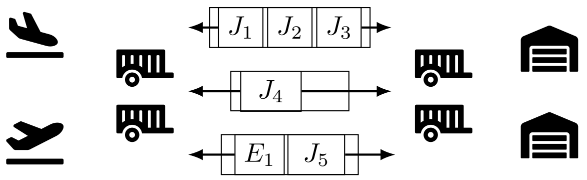

Figure 1: The Beluga AI Challenge setting.

1. Why is jig X loaded on rack A instead of another rack B?
2. Why load jig B on rack D instead of loading jig C on rack A?
3. Why not load jig X on rack A before loading jig Y on rack D?
4. How can I reduce the number of swaps?
5. What is the impact of removing rack A for maintenance?
6. How can I keep one rack empty all the time?

In these queries, X, A, B, C, and D are variables that are replaced with specific jigs and racks. Note that a rack removed for maintenance cannot have anything new jig placed on it, but the jigs currently on it can still be accessed as normal.

We do not attempt to answer question type 4 (swap reductions), which boils down to finding optimal plans. Neither do we tackle the setting where the input task is unsolvable.

Framework Overview

We aim to answer questions such as those posed in the Beluga challenge by generating counterfactuals, plans that adhere to additional question-specific constraints, and then comparing them to the original plan. To accomplish this, we takes as input a planning task, an initial plan and a query before performing three steps to come up with an explanation:

1. **Reformulation**: The original task is modified to create a new task which, when solved, produces a counterfactual plan. Each question has a specific reformulation, described in the next section.
2. **Solving**: The reformulated task is solved using a planner. In the competition, the anytime variant of 2011 LAMA (Richter and Westphal 2010; Richter, Westphal, and Helmert 2011) within the Unified Planning Framework (Micheli et al. 2025) is used, which is implemented in Fast Downward (Helmert 2006). However, any planner that can solve the reformulated task can be used, with optimal plans being preferred to give the best explanations.
3. **Explaining**: A large language model (LLM) is used to generate a natural-language explanation by comparing the original and counterfactual plan. For details about the prompt and the LLM, see section "LLM Explanations".

Reformulations Based on Queries/Questions

Given a user query, we aim to produce a counterfactual plan. We focus on the specific queries posed in the Beluga challenge, but they can be easily be generalized to *any* planning task. In the following, we briefly outline the reformulations used for each of the questions.

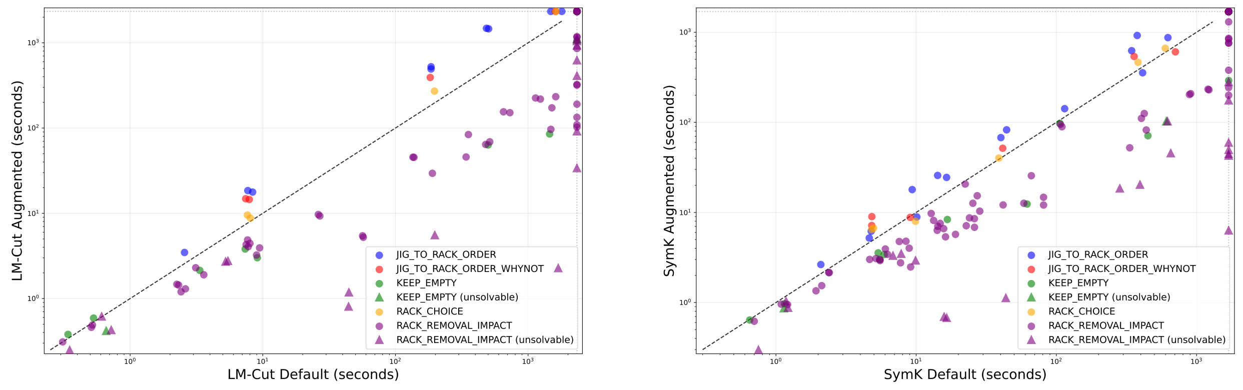

Figure 2: Performance of LM-Cut and SymK on the original and reformulated tasks for the Beluga Explainability benchmarks.

Why do *this* instead of *that*? In this query, the user is wondering why, at some point in the plan, a specific ground atom is reached, when they think a better plan could be achieved if another specified ground atom were reached at some point instead. To find this alternate plan, we create a planning task where we forbid the original ground atom from ever being true, and we force the alternate ground atom to be true at some point in every plan.

Formally, given a task $\Pi = \langle D, I \rangle$, and two ground atoms $p(t)$ and $q(r)$, where $p, q \in \mathcal{P}$ are predicates, $t = \langle t_1, \ldots, t_n \rangle$ and $r = \langle r_1, \ldots, r_m \rangle$ are tuples of typed constants, we create the modified task $\Pi^{\text{instead}}_{p(t),\,q(r)} = \langle D', I' \rangle$, $D' = \langle \mathcal{H}, \mathcal{C}, \mathcal{P}', \mathcal{A}' \rangle$, $I' = \langle \mathcal{O}, \mathcal{I}', \mathcal{G}' \rangle$, where:

- The type hierarchy $\mathcal{H}$ remains unchanged.
- The set of domain constants $\mathcal{C}$ remains unchanged.
- $\mathcal{P}' := \mathcal{P} \cup \{\text{forbidden-p}, \text{was-true-q}\}$, where forbidden and was-true have the same arity and argument types as p and q, respectively.
- $\mathcal{A}' := \{a' \mid a \in \mathcal{A}\}$, where there is one action schema a' for each action schema a in the original task, defined as $a' := \langle name(a), params(a), pre^+(a), pre^-(a) \cup \alpha, add(a) \cup \beta, del(a), cost(a) \rangle$, where $\alpha := \{\text{forbidden-p}(x) \mid p(x) \in add(a)\}$, while $\beta := \{\text{was-true-q}(y) \mid q(y) \in add(a)\}$.
- The set of typed constants $\mathcal{O}$ remains unchanged.
- The initial state forbids the ground atom $p(t)$: $\mathcal{I}' := \mathcal{I} \cup \{\text{forbidden-p}(t)\}$.
- The goal requires the ground atom $q(r)$ to be true at some point in the plan: $\mathcal{G}' := \mathcal{G} \cup \{\text{was-true-q}(r)\}$.

With this formulation, no plan for $\Pi^{\text{instead}}_{p(t),\,q(r)}$ will reach the ground atom $p(t)$ (since any action that achieves it will not be applicable) and all will contain the ground atom $q(r)$ at some point, thus providing a counterfactual plan.

In the Beluga explainability challenge, this reformulation corresponds to two different queries: *why is jig X loaded on rack A instead of another rack B?*, and *why load jig B on rack D instead of loading jig C on rack A?*

Note that $\Pi^{\text{instead}}_{p(t),\,q(r)}$ can easily be extended to forbid or enforce sets of atoms, simply by adding the corresponding forbidden-x and was-true-y predicates. In fact, by forbidding a set of ground atoms, we solve a different Beluga challenge query: *what is the impact of removing a rack for maintenance?* When a rack is on maintenance, it can still be unloaded, but no new jigs can be loaded onto it. For this, we simply forbid all ground atoms that state that a jig is loaded on that specific rack.

Why this *before* that? In this query, the user is wondering why a specific ground atom $q(r)$ is reached in the plan before another specific ground atom $p(t)$. To find a counterfactual plan in this case, we reformulate the task to guarantee two things: first, that the ground atom $p(t)$ cannot be reached before $q(r)$, and second, that both ground atoms must be true at some point in the plan. This enforces the counterfactual ordering to the question.

For task $\Pi = \langle D, I \rangle$, and two ground atoms $p(t)$ and $q(r)$ for which we want to enforce that $p(t)$ happens before $q(r)$, we create the modified task $\Pi^{\text{order}}_{p(t),\,q(r)} = \langle D', I' \rangle$, $D' = \langle \mathcal{H}, \mathcal{C}, \mathcal{P}', \mathcal{A}' \rangle$, $I' = \langle \mathcal{O}, \mathcal{I}', \mathcal{G}' \rangle$, where:

- $\mathcal{H}$, $\mathcal{C}$ and $\mathcal{O}$ remain unchanged.
- $\mathcal{P}' := \mathcal{P} \cup \{\text{was-true-p}, \text{was-true-q}, \text{forbidden-q}, \text{needed-p}\}$.
- $\mathcal{A}' := \{a' \mid a \in \mathcal{A}, q \notin add(a)\} \cup \{a'', a''' \mid a \in \mathcal{A}, q(x) \in add(a)\}$. We create one action schema a' for each action schema that does not include predicate q in its add list: $a' := \langle name(a), params(a), pre^+(a), pre^-(a), add(a) \cup \beta, del(a), cost(a) \rangle$, with $\beta := \{\text{was-true-p}(x) \mid p(x) \in add(a)\}$. Additionally, we create two action schemas for each action schema that includes q in its add list:

  $a'' := \langle name(a), params(a), pre^+(a), pre^-(a) \cup \{\text{forbidden-q}(x)\}, add(a), del(a), cost(a) \rangle$, with negative precondition forbidden-q(x) ensuring that the corresponding ground action does not make the ground atom $q(r)$ true before $p(t)$ was true.

  $a''' := \langle name(a), params(a) \cup y, pre^+(a) \cup \{\text{forbidden-q}(x), \text{needed-p}(y), \text{was-true-p}(y)\}, pre^-(a), add(a) \cup \{\text{was-true-q}(x)\}, del(a), cost(a) \rangle$, where was-true-p$(y)$ ensures that $p(t)$ was true before executing an action that adds $q(r)$, and y are the parameters of p, that we need to introduce to the action schema to ensure $p(y)$ matches $p(t)$.

- The initial state forbids ground atom $q(r)$ and sets $p(t)$ as needed: $\mathcal{I}' := \mathcal{I} \cup \{\text{forbidden-q}(r), \text{needed-p}(t)\}$.
- The goal requires both $p(t)$ and $q(r)$ to be true at some point in the plan: $\mathcal{G}' := \mathcal{G} \cup \{\text{was-true-p}(t), \text{was-true-q}(r)\}$.

$\Pi^{\text{order}}_{p(t),\, q(r)}$ is used to answer the Beluga query *why (not) load jig X on rack A before loading jig Y on rack D?*

Can you *not* use that object? Here, the user is wondering if you can solve the planning task without the use of a specific object. In this case, to reformulate the task to get a counterfactual plan, we simply remove said object from the set of objects, and adapt the initial state accordingly. This corresponds to the Beluga query *how can I keep one rack empty all the time?* For this specific case, we take the *smallest* rack that starts empty, and remove it from the task.

LLM Explanations

We use the large language model (LLM) o3-mini[2] to generate natural language explanations for planning decisions in the Beluga planning problem. The LLM is given structured input describing the objectives, constraints, and trade-offs involved in the planning process along with one example. See the appendix for the full prompt.

Together with the static prompt, the model receives specific scenario information in the form of both the original and counterfactual plans, together with the costs associated with both, and the question asked. The model is tasked with determining the reasoning behind the question-related actions and highlighting trade-offs between the plans to provide an explanation for both the question and the effects caused in the modified plans.

Experimental Evaluation

We experimentally evaluate the performance impact of our reformulations. Additionally, we showcase and discuss an example answer generated by our system.

Reformulation Difficulty

To empirically evaluate the impact our task modifications have, we use the satisfiable benchmark set from the Beluga Explainability Challenge filtered to those with questions supported by our framework, i.e., the request to reduce swaps has been excluded. We use regular settings for the international planning competition (IPC), i.e., 8 GB memory limit and 30 minutes time limit, and compare the runtime for solving the original and reformulated tasks for two state-of-the-art optimal planners, LM-Cut (Helmert and Domshlak 2009) implemented in Fast Downward (Helmert 2006) and SymK (Speck, Mattmüller, and Nebel 2020). The results are shown in Figure 2 and Table 1.

We see that SymK outperforms LM-Cut across the board, solving 99 of the 224 original tasks and 115 of the reformulated tasks, while LM-Cut only solves 67 and 81 respectively. Additionally, for SymK, we see that any original task

[2]https://openai.com/index/openai-o3-mini/

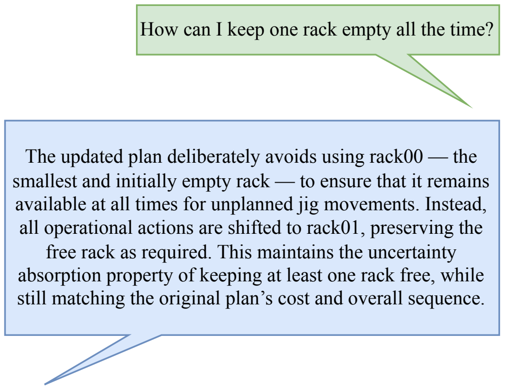
How can I keep one rack empty all the time?
The updated plan deliberately avoids using rack00 — the smallest and initially empty rack — to ensure that it remains available at all times for unplanned jig movements. Instead, all operational actions are shifted to rack01, preserving the free rack as required. This maintains the uncertainty absorption property of keeping at least one rack free, while still matching the original plan's cost and overall sequence.

Figure 3: An example interaction with our system.

that can be solved can also be solved in its reformulated version, while LM-Cut fails to solve 4 reformulated tasks that it can solve in the original version. Conversely, we see that both planners have cases where they can solve the reformulated task but not the original task, 16 for SymK and 18 for LM-Cut. As such, we see an average speedup factor of 3.45 for LM-Cut and 2.08 for SymK, highlighting that the reformulated tasks are usually easier to solve.

Looking at the question types, we see that those which remove racks are notably easier to solve, while those that consider jig choices instead usually grow marginally harder. The only tasks proven unsolvable are those where racks are removed, specifically those which start with only two racks and retain only a single remaining rack after the removal.

Answer Quality

Figure 3 shows an example interaction with our system for a Beluga Explainability query. The answer describes both how the modified plan achieves the goal, re-routing to rack01, and the reasoning behind the modification, keeping rack00 empty as it was the smallest initially empty rack. Additionally, it reasons about why one might wish to keep a rack empty, namely to allow for dealing with real-world uncertainties. However, it also assumes that the user has access to the modified plan. Analyzing said plan directly might be difficult for an untrained user, and as such pairing the high-level explanations and motivations from our system with a visualisation software or similar tool would be beneficial.

We see similar behaviour for other questions. However, the fact that most initial plans in the competition were suboptimal means that our modified plans frequently outperform the originals. This, in turn, often highlights original suboptimalities rather than identifying difficulties in the original task, but is nevertheless useful for analyzing, understanding and critiquing the original plans.

Related Work

Since counterfactual reasoning or task reformulations for insight and explainability are common in the field of explainable AI planning, we can only highlight the most relevant examples for this work. Göbelbecker et al. (2010) reformulate

		Jig to Rack Order	Why Not	Keep Empty	Rack Choice	Rack Removal	Total
LM-Cut	Original	9	4	8	4	42	67
	Reformulated	7	3	10	3	58	81
SymK	Original	14	6	11	6	62	99
	Reformulated	14	6	12	6	77	115
	Total	50	33	12	33	96	224

Table 1: Number of problems solved or proven unsolvable for original and reformulated tasks with LM-Cut and SymK.

the task to allow for arbitrary changes to the initial state before planning, to allow for explanations of unsolvable tasks with a focus on the initial state. The works of Gragera et al. (2023) and Gragera and Muise (2024) do task reformulations on the actions to explain unsolvable tasks. The approach of Cashmore et al. (2019) and Krarup et al. (2019) is most similar to ours and allows for general, question-adapted reformulations of the task resulting in contrastive explanations. Their reformulations are slightly different, allowing for durative actions and temporal constraints.

Adding constraints to an existing planning problem is officially supported with PDDL 3 (Gerevini and Long 2005) and would allow for a simpler expression of our reformulations, but most planners do not support them, so they are usually compiled away (Edelkamp 2006). Other reformulations are used for performance reasons, for example, macro-operators (Botea et al. 2005), bagging (Riddle et al. 2016), and action schema splitting (Alarnaouti, Baryannis, and Vallati 2023).

Contrastive analysis can be extended to not only compute one counterfactual plan but explore the family of counterfactual plans as shown by Eifler et al. (2020).

An alternative approach to come up with explanations is based on facet reasoning (Gnad et al. 2025).

Conclusion

We propose an approach to explainable AI planning (XAIP) based on counterfactual reasoning. The key component of our system are query-specific task reformulations that provide evidence for the consequences of user-desired alternatives. While the reformulations are inspired by the specific queries ansked in the Beluga Challenge, we introduce a general framework that is applicable to any planning task. Connecting the symbolic information obtained by solving the modified task to a large language model allows our system to be used by lay persons not familiar with planning technicalities. We consider complementing our textual explanations with a domain-specific visualization as a promising improvement to further increase the value of the system.

References

Alarnaouti, D.; Baryannis, G.; and Vallati, M. 2023. Reformulation techniques for automated planning: a systematic review. *The Knowledge Engineering Review*, 38: e9.

Botea, A.; Enzenberger, M.; Müller, M.; and Schaeffer, J. 2005. Macro-FF: Improving AI Planning with Automatically Learned Macro-Operators. *JAIR*, 24: 581–621.

Cashmore, M.; Collins, A.; Krarup, B.; Krivić, S.; Magazzeni, D.; and Smith, D. 2019. Towards Explainable AI Planning as a Service. In *ICAPS 2019 Workshop on Explainable AI Planning (XAIP)*.

Corrêa, A. B.; Pommerening, F.; Helmert, M.; and Francès, G. 2020. Lifted Successor Generation using Query Optimization Techniques. In *Proc. ICAPS 2020*, 80–89.

Edelkamp, S. 2006. On the Compilation of Plan Constraints and Preferences. In *Proc. ICAPS 2006*, 374–377.

Eifler, R.; Cashmore, M.; Hoffmann, J.; Magazzeni, D.; and Steinmetz, M. 2020. A New Approach to Plan-Space Explanation: Analyzing Plan-Property Dependencies in Oversubscription Planning. In *Proc. AAAI 2020*, 9818–9826.

Gerevini, A. E.; and Long, D. 2005. Plan Constraints and Preferences in PDDL3. Technical Report R. T. 2005-08-47, University of Brescia, Department of Electronics for Automation.

Gnad, D.; Hecher, M.; Gaggl, S.; Rusovac, D.; Speck, D.; and Fichte, J. K. 2025. Interactive Exploration of Plan Spaces. In *Proc. KR 2025*.

Göbelbecker, M.; Keller, T.; Eyerich, P.; Brenner, M.; and Nebel, B. 2010. Coming Up With Good Excuses: What to do When no Plan Can be Found. In *Proc. ICAPS 2010*, 81–88.

Gragera, A.; Fuentetaja, R.; Olaya, A.; and Fernández, F. 2023. PDDL Domain Repair: Fixing Domains with Incomplete Action Effects. ICAPS 2023 System Demonstrations.

Gragera, A.; and Muise, C. 2024. One Repair to Rule Them All: Repairing a Broken Planning Domain Using Multiple Instances. In *ICAPS 2025 Workshop on Workshop on Reliable Data-Driven Planning and Scheduling (RDDPS)*.

Helmert, M. 2006. The Fast Downward Planning System. *JAIR*, 26: 191–246.

Helmert, M.; and Domshlak, C. 2009. Landmarks, Critical Paths and Abstractions: What's the Difference Anyway? In *Proc. ICAPS 2009*, 162–169.

Krarup, B.; Cashmore, M.; Magazzeni, D.; and Miller, T. 2019. Model-based contrastive explanations for explainable planning. In *ICAPS 2019 Workshop on Explainable AI Planning (XAIP)*.

Micheli, A.; Bit-Monnot, A.; Röger, G.; Scala, E.; Valentini, A.; Framba, L.; Rovetta, A.; Trapasso, A.; Bonassi, L.; Gerevini, A. E.; Iocchi, L.; Ingrand, F.; Köckemann, U.; Patrizi, F.; Saetti, A.; Serina, I.; and Stock, S. 2025. Unified Planning: Modeling, manipulating and solving AI planning problems in Python. *SoftwareX*, 29: 102012.

Richter, S.; and Westphal, M. 2010. The LAMA Planner: Guiding Cost-Based Anytime Planning with Landmarks. *JAIR*, 39: 127–177.

Richter, S.; Westphal, M.; and Helmert, M. 2011. LAMA 2008 and 2011 (planner abstract). In *IPC 2011 Planner Abstracts*, 50–54.

Riddle, P.; Douglas, J.; Barley, M.; and Franco, S. 2016. Improving Performance by Reformulating PDDL into a Bagged Representation. In *ICAPS 2016 Workshop on Heuristics and Search for Domain-independent Planning*, 28–36.

Speck, D.; Mattmüller, R.; and Nebel, B. 2020. Symbolic Top-k Planning. In *Proc. AAAI 2020*, 9967–9974.

Tuples. 2025. Beluga™ AI Challenge. https://tuples.ai/competition-challenge/.

Appendix - LLM Prompt

Setting

Consider the following properties grouped by objective:

Plan Quality Metric:

- **Minimum number of swaps:** A swap occurs when a jig is moved between two racks, temporarily stored in a trailer, or placed back on the same rack to gain access to otherwise blocked jigs.

Uncertainty Absorption Properties:

- Keeping at least one rack free.
- Placing shorter jigs on shorter racks.
- Maintaining an empty rack allows unplanned jig movements between the Beluga hangar and the factory without prior rearrangement.

Facilitating Manual Planning:

- Placing jigs of the same type on the same rack.
- Storing empty jigs on the Beluga side of the rack system.
- Storing full jigs on the factory side of the rack system.

Since these properties conflict, a trade-off must be achieved.

Cases Considered:

1. **Infeasible Problem Instance:**
   - The human planner needs a clear, concise explanation of why the instance is infeasible.
   - Suggestions should be provided on how feasibility could be restored.
2. **Solvable Problem Instance:**
   - If a reference plan exists, alternative solutions proposed by the human planner must be justified.
   - Explanations should compare the reference plan with proposed alternatives and outline their trade-offs.

Even when a problem is feasible and a reference plan is provided, explanations are still required. The human planner may have expected a different approach or a different trade-off. For example, a reference plan may keep one rack empty but require 10 swaps. A planner may then ask: *"How can I reduce the number of swaps?"*

Expectations for Explanations:

- Determine the feasibility of an alternative solution.
- Provide insight into the consequences of the proposed alternative.
- Compare the chosen plan with alternative scenarios and explain why specific choices were made.

Task

You are an AI system rewriting planner outputs into natural language explanations. Your goal is to maximize explanation quality based on the above criteria, ensuring the response is useful and understandable to domain experts unfamiliar with AI-driven planning. One of the key challenges in addressing the Beluga planning problem is ensuring that the techniques used are explainable. You must demonstrate that their methods provide clear and insightful explanations for how and why decisions were made.

Provide the explanation directly, using markdown formatting to enhance readability.

Important Considerations:

- The original plan was generated using a suboptimal planner.
- The new plan was generated using an optimization-focused planner.
- The original plan is not necessarily optimal and does not explicitly optimize for factors like swaps.
- The new plan optimizes for plan length while answering the question but does not optimize for other metrics.
- There might be no clear anwser to why a certain action was made, even tho the question asked about this specific action
- Focus on the action in foucs and the surrounding actions when answering the question
- The explanation should be short and concise, one paragraph

Explanation: Provide a clear, structured answer that explains the reasoning behind the plan modifications, trade-offs, and implications. The explanation should be short and concise, and be focused on the question that is asked.

Example:

Example Input: Original plan: SequentialPlan: pick-up-rack(jig0002, factory-trailer-1, rack00, fside) unload-beluga(jig0003, beluga1, beluga-trailer-1) switch-to-next-beluga() put-down-rack(jig0003, beluga-trailer-1, rack00, bside) pick-up-rack(jig0003, factory-trailer-2, rack00, fside) deliver-to-hangar(jig0003, hangar1, factory-trailer-2, pl0) deliver-to-hangar(jig0002, hangar2, factory-trailer-1, pl1) get-from-hangar(jig0002, hangar2, factory-trailer-1) put-down-rack(jig0002, factory-trailer-1, rack00, fside) pick-up-rack(jig0001, factory-trailer-2, rack01, fside) deliver-to-hangar(jig0001, hangar2, factory-trailer-2, pl2) get-from-hangar(jig0001, hangar2, factory-trailer-2) put-down-rack(jig0001, factory-trailer-2, rack01, fside) pick-up-rack(jig0001, beluga-trailer-1, rack01, bside) load-beluga(jig0001, beluga2, beluga-trailer-1) switch-to-next-beluga() unload-beluga(jig0004, beluga3, beluga-trailer-1) put-down-rack(jig0004, beluga-trailer-1, rack00, bside) unload-beluga(jig0005, beluga3, beluga-trailer-1) put-down-rack(jig0005, beluga-trailer-1, rack01, bside) unload-beluga(jig0006, beluga3, beluga-trailer-1) put-down-rack(jig0006, beluga-trailer-1, rack01, bside) pick-up-rack(jig0005, factory-trailer-1, rack01, fside) pick-up-rack(jig0006, factory-trailer-2, rack01, fside) deliver-to-hangar(jig0006, hangar2, factory-trailer-2, pl2) pick-up-rack(jig0002, factory-trailer-2, rack00, fside) put-down-rack(jig0002, factory-trailer-2, rack01, fside) pick-up-rack(jig0004, factory-trailer-2, rack00, fside) pick-up-rack(jig0002, beluga-trailer-1, rack01, bside) load-beluga(jig0002, beluga3, beluga-trailer-1) put-down-rack(jig0005, factory-trailer-1, rack00, fside) get-from-hangar(jig0003, hangar1, factory-trailer-1) deliver-to-hangar(jig0004, hangar1, factory-trailer-2, pl1) pick-up-rack(jig0005, factory-trailer-2, rack00, fside) put-down-rack(jig0003, factory-trailer-1, rack00, fside) pick-up-rack(jig0003, beluga-trailer-1, rack00, bside) load-beluga(jig0003, beluga3, beluga-trailer-1) switch-to-next-beluga() get-from-hangar(jig0004, hangar1, factory-trailer-1) deliver-to-hangar(jig0005, hangar1, factory-trailer-2, pl0) put-down-rack(jig0004, factory-trailer-1, rack00, fside) pick-up-rack(jig0004, beluga-trailer-1, rack00, bside) load-beluga(jig0004, beluga4, beluga-trailer-1) get-from-hangar(jig0005, hangar1, factory-trailer-2) put-down-rack(jig0005, factory-trailer-2, rack00, fside) pick-up-rack(jig0005, beluga-trailer-1, rack00, bside) load-beluga(jig0005, beluga4, beluga-trailer-1)

Original cost: 47

Question: Why put jig jig0004 in rack rack00 before loading jig jig0006 in rack rack01?

Plan fulfilling the question (i.e. where jig jig0006 is placed on rack rack01 before jig jig0004 is placed on rack rack00): SequentialPlan: pick-up-rack(jig0002, factory-trailer-1, rack00, fside, bside, n11, n11, n22) unload-beluga(jig0003, dummy-jig, beluga-trailer-1, beluga1) beluga-complete(beluga1, beluga2) put-down-rack-modified(jig0003, beluga-trailer-1, rack00, bside, n11, n22, n11) pick-up-rack(jig0003, factory-trailer-2, rack00, fside, bside, n11, n11, n22) deliver-to-hangar(jig0003, jig0005, factory-trailer-2, hangar1, pl0, n11, n08) deliver-to-hangar(jig0002, jig0004, factory-trailer-1, hangar2, pl1, n11, n08) get-from-hangar(jig0002, hangar2, factory-trailer-1) pick-up-rack(jig0001, factory-trailer-2, rack01, fside, bside, n32, n01, n33) put-down-rack-modified(jig0002, factory-trailer-1, rack00, fside, n08, n22, n14) deliver-to-hangar(jig0001, jig0006, factory-trailer-2, hangar2, pl2, n32, n32) get-from-hangar(jig0001, hangar2, factory-trailer-2) put-down-rack-modified(jig0001, factory-trailer-2, rack01, fside, n32, n33, n01) pick-up-rack(jig0001, beluga-trailer-1, rack01, bside, fside, n32, n01, n33) load-beluga(jig0001, typee, dummy-type, beluga2, beluga-trailer-1, slot0, dummy-slot) beluga-complete(beluga2, beluga3) unload-beluga(jig0004, jig0005, beluga-trailer-1, beluga3) put-down-rack-modified(jig0004, beluga-trailer-1, rack01, bside, n11, n33, n22) unload-beluga(jig0005, jig0006, beluga-trailer-1, beluga3) stack-rack-modified(jig0005, jig0004, beluga-trailer-1, rack01, bside, fside, n11, n22, n11) unload-beluga(jig0006, dummy-jig, beluga-trailer-1, beluga3) stack-rack-jig-b-rack-b(jig0006, jig0005, beluga-trailer-1, rack01, bside, fside, n11, n11, n00) unstack-rack(jig0004, jig0005, factory-trailer-1, rack01, fside, bside, n11, n00, n11) unstack-rack(jig0005, jig0006, factory-trailer-2, rack01, fside, bside, n11, n11, n22) pick-up-rack(jig0002, beluga-trailer-1, rack00, bside, fside, n08, n14, n22) deliver-to-hangar(jig0005, dummy-jig, factory-trailer-2, hangar2, pl0, n11, n08) get-from-hangar(jig0003, hangar1, factory-trailer-2) deliver-to-hangar(jig0004, dummy-jig, factory-trailer-1, hangar1, pl1, n11, n08) load-beluga(jig0002, typeb, typeb, beluga3, beluga-trailer-1, slot0, slot1) pick-up-rack(jig0006, factory-trailer-1, rack01, fside, bside, n11, n22, n33) put-down-rack-modified(jig0003, factory-trailer-2, rack00, fside, n08, n22, n14) get-from-hangar(jig0004, hangar1, factory-trailer-2) deliver-to-hangar(jig0006, dummy-jig, factory-trailer-1, hangar1, pl2, n11, n08) stack-rack-jig-x-rack-a(jig0004, jig0003, factory-trailer-2, rack00, fside, bside, n08, n14, n06) unstack-rack(jig0003, jig0004, beluga-trailer-1, rack00, bside, fside, n08, n06, n14) load-beluga(jig0003, typeb, dummy-type, beluga3, beluga-trailer-1, slot1, dummy-slot) beluga-complete(beluga3, beluga4) pick-up-rack(jig0004, beluga-trailer-1, rack00, bside, fside, n08, n14, n22) load-beluga(jig0004, typeb, typeb, beluga4, beluga-trailer-1, slot0, slot1) get-from-hangar(jig0005, hangar2, factory-trailer-2) put-down-rack-modified(jig0005, factory-trailer-2, rack00, fside, n08, n22, n14) pick-up-rack(jig0005, beluga-trailer-1, rack00, bside, fside, n08, n14, n22) load-beluga(jig0005, typeb, dummy-type, beluga4, beluga-trailer-1, slot1, dummy-slot)

Cost for fulfilling plan: 43

Example Output: There is no specific reason to why to put jig jig0004 in rack rack00 before loading jig jig0006 in rack rack01. The new plan removes this action and creates a plan with lower cost.

To Explain

{Current task to explain is placed here}