Mojified Pacman: A Deterministic and Fully Observable Variant for PDDL Modeling Competitions

Bruno Ribas, Igor Penha, Lucas Bergholz, Bruno Ribeiro

University of Brasília,
Brasília, DF – Brazil
bruno.ribas@unb.br, igorpenha.it@gmail.com, lucas.bergholz@gmail.com, bbrunoo@icloud.com

Abstract

In this work, we address a challenging planning problem inspired by the classic arcade game Pacman, reimagined as a deterministic and fully observable turn-based variant. The objective is to eliminate adversaries and navigate a grid environment enriched with dynamic elements such as teleportation portals, ice tiles, and collectible fruits that affect agent capabilities. The game's mechanics involve asynchronous movement, conditional interactions, and spatial reasoning, making it a complex but natural candidate for PDDL modelling. While the problem can be expressed using standard PDDL features, its intricacies reveal modelling challenges that impact plan optimality and solver performance. We propose a benchmark composed of procedurally generated maps with varying combinations of terrain features and difficulty levels, classified into three competition tracks: agile, satisficing, and optimal. This work offers a new, expressive domain for evaluating the capabilities of planning systems and raises important questions about the trade-offs between modelling precision and solving efficiency.

Introduction

Designing planning problems based on video games presents a compelling opportunity to evaluate the expressiveness and efficiency of planning tools in highly dynamic environments. Grid-based games, in particular, which incorporate movement, conditional interactions, and goal-driven behaviour, naturally lend themselves to planning representations. However, complexity can increase significantly with the introduction of multiple agents, rule-based adversaries, and varying terrains.

Among the most prominent tools for modelling such problems is the Planning Domain Definition Language (PDDL) (McDermott 2000), whose high-level abstraction allows a concise representation of fundamental domain components. Its adoption was largely consolidated through the International Planning Competition (IPC), held within the International Conference on Automated Planning and Scheduling (ICAPS), which established PDDL as a standard for evaluating planning systems.

In this work, we explore a modified version of the classic arcade game Pacman (Iwatani 1980), adapted for AI planning and implemented on the CD-MOJ platform.[1] Our turn-based variation, titled *Mojified* Pacman, incorporates several strategic elements such as diverse ghost behaviours, collectible power-ups (fruits), teleportation portals, and ice tiles that alter movement mechanics. These additions enhance the domain's expressiveness and introduce new challenges for modelling and resolution. To better understand these dynamics, Figure 1 provides an example map with key elements such as walls, ice tiles, and portals, illustrating the complexity and depth of this enhanced game environment.[2]

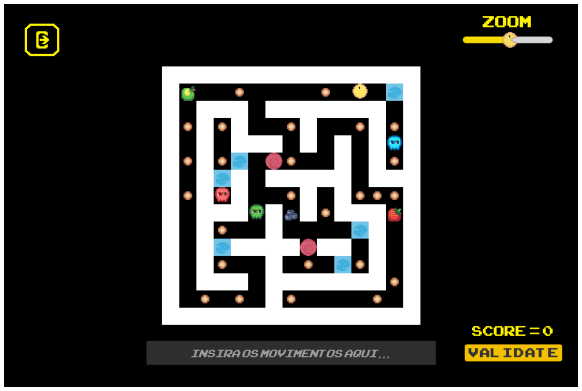

Figure 1: *Mojified* Pacman.

The benchmark aims to achieve two primary goals: firstly, to provide a rich environment for evaluating planning models in terms of correctness, optimality, and efficiency; and secondly, to investigate the trade-offs between expressive PDDL modelling and planner performance across various competition tracks — agile (speed-focused), satisficing (cost-aware), and optimal (minimal-cost). Like Sokoban (Imabayashi 1982) and other path-based puzzle games, modelling reachability, interactions, and dynamic rules is crucial, especially when agents move asynchronously and conditionally affect one another.

[1] https://moj.naquadah.com.br.

[2] https://icebergholz.itch.io/pacman-mojificado.

By constructing a PDDL domain and conducting a structured evaluation on procedurally generated maps, we analyse the impact of different configurations on a planner's ability to find valid or optimal solutions within specific time ranges. This paper introduces a new game-inspired benchmark for the planning community and sheds light on modelling and reasoning within richly dynamic domains.

The Game

For this modified version of Pacman, called *Mojified* Pacman (pun with the platform that the game was launched), the objective of the game will be altered. According to the track competed for by the participant, the objective for each competition track will be:

- **Agile**: defeat all the ghosts as quickly as possible;
- **Satisficing**: defeat all the ghosts and collect as many pellets as possible in the meantime, to achieve the lowest cost of movements possible;
- **Optimal**: eliminate all the ghosts with the lowest cost of movements.

As the game is not in real time like the original, it will be divided into turns. During each turn, Pacman can move one cell, after which all ghosts will make their moves, allowing Pacman to move again. The player is always in motion, meaning that regardless of collecting a fruit or pellet, or colliding with a ghost, Pacman must always move to another location. In case the player tries to move in the direction of a wall, Pacman will remain in the same cell, while the ghosts will make their movements.

Additionally, unlike the original game, there are only three types of ghosts, each of them following a set of deterministic movement patterns:

- **Red**: moves in a clockwise direction, changing its movement direction when it encounters a wall. It starts moving to the right until it collides with a wall, then moves down, then left, then up. This ghost never stops moving; if it encounters a wall, it tries to move in the next direction;
- **Green**: mimics Pacman's movements. If Pacman moves down, this ghost moves down; if he moves left, the ghost moves left, and so on;
- **Blue**: moves opposite to Pacman's movements. If Pacman moves down, this ghost moves up; if he moves left, the ghost moves right, and so on.

Furthermore, each ghost has a corresponding fruit of the same colour. When the player eats one of these fruits, they can eliminate the corresponding ghost at any time during the game, unlike in the original game where this was only possible for a limited time. It is important to note that the player cannot have two active fruits simultaneously. For example, if the player eats the red fruit (enabling them to eliminate the red ghost), and then eats the blue fruit a few moves later, they will only be able to eliminate the blue ghost from that point onward.

Another addition to this version of Pacman is the inclusion of different types of flooring:

- **Normal Floor**: Represented by a space character, this is a standard type of floor with no special features. Underneath every ghost and Pacman, at the moment they appear in the game, there is this type of floor;
- **Ice Floor**: Pacman slides across this type of floor in the direction he enters, continuing until it reaches a cell without ice. However, if he encounters a wall along the way, he will collide with it and then reverse direction until he exits the ice;
- **Portal**: The portal floor acts as a link to another point on the map. There can only be a single pair of portal cells, or no pair at all. If Pacman enters a portal, he will exit at the other portal cell.

As previously mentioned, both the Satisficing and Optimal tracks implement action-cost, meaning that each possible movement made by the player, the corresponding cost of it will be added to the total score of movements. All possible movements increase cost, but, some will increase more than others as detailed in Table 1:

Table 1: Costs per Movement.

Movement	**Cost**
Normal movement	2
Any fruit is activated	4
Moves into pellet (without any fruit)	1
Moves into wall without fruit	4
Moves into wall with fruit	8

With the modifications made to the original game, some border cases appear when you combine the new floors and mechanics. To clear up some of these cases:

- When a player walks into a portal, the cost of the movement will be as if he walked into a normal floor, even though he gets teleported;
- When a player walks over ice, for each tile he passes, the cost increases according to the cost of moving into a normal tile (2 if no fruit is activated or 4 if the player has a fruit equipped);
- If the player hits a wall while sliding on ice, the cost of hitting it isn't applied. This happens because the cost is applied only when the player purposely moves into a wall.

PDDL Formulation

While the domain exhibits structural similarities to Sokoban, particularly in its grid based movement and spatial reasoning, it significantly diverges in both complexity and objectives. In contrast, the formulation presented in this work introduces dynamic elements that extend far beyond Sokoban's scope. It incorporates multiple agents, including Pacman and ghosts, each governed by their own movement and behavioral rules. The presence of special types of floors — such as icy tiles that affect directional movement and portals that enable teleportation - further adds to the complexity of the domain. Furthermore, the domain includes combat-like mechanics through

predicates such as check-kill and check-dead, which govern the temporal logic of interactions between Pacman and ghosts.

The model for Pacman follows a sequential turn-based structure and is defined using PDDL version 2.1 (Fox and Long 2003). Each turn begins with Pacman's movement. After Pacman moves, the system checks whether he has eliminated any ghosts. Subsequently, it verifies whether Pacman has been eliminated by any ghost. If Pacman's new position contains a fruit, he collects it, and the effect of the fruit remains active until he either eliminates the corresponding ghost or acquires a different fruit.

Following Pacman's turn, control passes to the ghosts. Each ghost executes its movement individually. After all ghosts have moved, the model reevaluates whether, based on their new positions, Pacman has either eliminated or been eliminated by a ghost. Once this evaluation is complete, the next turn begins with Pacman again.

Types, objects and fluents

In this domain, no explicit types are defined beyond positional coordinates. The problem is modeled entirely over a two-dimensional Cartesian grid.

- **Adjacency relations**:
  - `(inc ?a ?b)`, `(dec ?a ?b)` establish successor and predecessor links between coordinates.
- **Environment features**:
  - `(wall ?x ?y)`, `(dot ?x ?y)`, `(ice ?x ?y)`, `(tele ?x ?y)` characterize special grid cells such as walls, collectible dots, slippery tiles, and teleportation portals.
- **Entity positions**:
  - `(enemyX-at ?x ?y)`, `(fruitX-at ?x ?y)`, `(player-at ?x ?y)` represent the locations of ghosts, fruits, and Pacman (**X** $\in$ **{B, G, R}**).
- **Game-flow auxiliaries**:
  - `(has-action)`, `(check-kill)`, `(check-dead)`, `(check-from-enemy)`, `(get-fruit)` encode auxiliary conditions for turn coordination, combat outcomes, and fruit collection.
- **State markers**:
  - `(is-dead)`, `(deadX)` register whether Pacman or a specific ghost has been eliminated (**X** $\in$ **{B, G, R}**).
- **Inventory markers**:
  - `(player-withX)` capture whether Pacman is currently carrying a fruit of type **X** $\in$ **{B, G, R}**.
- **Movement effects**:
  - `(iceD)`, `(iceU)`, `(iceL)`, `(iceR)`, `(telemove)` express special movements triggered by icy surfaces (with direction down, up, left, right) or teleportation tiles.
- **Enemy intentions**:
  - `(enemyX-up)`, `(enemyX-down)`, `(enemyX-left)`, `(enemyX-right)` specify the planned movement directions of each ghost (**X** $\in$ **{B, G, R}**).

Actions

The model defines a total of fifty-five actions; however, many of these are variations of a core set. The principal distinct actions include: `check_kill`, `check_dead`, `get_fruit`, `movement`, `movement_at_ice`, `teleportation`, `movement_ghosts`, and `movement_of_dead_ghost`. As indicated by their names, the `movement` action governs Pacman's movement from one tile to an adjacent tile, while the `movement_ghosts` action controls the movement of ghosts according to predefined behavioral rules. When a ghost is in a "dead" state during its turn, it executes the `movement_of_dead_ghost` action, which effectively results in skipping the turn without performing any additional operations.

The `check_kill` and `check_dead` actions determine whether Pacman has killed a ghost or has been killed by one. The logic prioritizes checking whether Pacman kills a ghost first, only afterward does the system verify whether another ghost has subsequently killed Pacman. These actions take the x and y coordinates of the game grid as parameters. For these actions to be executed, it must be the appropriate time step, and Pacman must be located at the specified (x, y) position.

When the `check_kill` action is executed, it iterates through all ghosts to determine if Pacman possesses the fruit that grants him the ability to eliminate a ghost of a specific color. If a ghost of that color is located on the same tile as Pacman, then the ghost's dead flag is set to true and the ghost ceases to exist on the grid. Conversely, the `check_dead` action verifies whether Pacman occupies the same tile as a ghost for which he lacks the corresponding fruit. If this condition is met, Pacman is considered dead.

Listing 1: Example of domain in PDDL.

```
(:action check_kill
 :parameters (?px ?py - position)
 :precondition (and (check-kill) (player-at ?px ?py))
 :effect (and
   (when (and (player-at ?px ?py) (enemyB-at ?px ?py)
              (player-withB))
         (and (not (enemyB-at ?px ?py)) (deadB)
              (not (player-withB))))
   (when (and (player-at ?px ?py) (enemyG-at ?px ?py)
              (player-withG))
         (and (not (enemyG-at ?px ?py)) (deadG)
              (not (player-withG))))
   (when (and (player-at ?px ?py) (enemyR-at ?px ?py)
              (player-withR))
         (and (not (enemyR-at ?px ?py)) (deadR)
              (not (player-withR))))
   (not (check-kill)) (check-dead)))

(:action check_dead
 :parameters (?px ?py - position)
 :precondition (and (check-dead) (player-at ?px ?py))
```

```
  :effect (and
    (when (and (player-at ?px ?py)
               (or (enemyB-at ?px ?py)
                   (enemyG-at ?px ?py)
                   (enemyR-at ?px ?py)))
          (is-dead))
    (when (and (not (check-from-enemy))
               (not (telemove))
               (not (geloC)) (not (geloB))
               (not (geloD)) (not (geloE)))
          (get-fruit))
    (when (check-from-enemy)
          (and (not (check-from-enemy))
               (not (has-action))))
    (when (and (deadR) (deadG) (deadB))
          (not (has-action)))
    (not (check-dead))))
```

When the check_dead signal is triggered by the ghosts after their respective movements, control is transferred to Pacman. However, if the check_dead event is initiated by Pacman and the current position does not correspond to a special floor tile, the get_fruit action must be executed. This action receives the x and y coordinates of Pacman's current position as parameters and verifies whether a fruit is present at that location. If a fruit exists, Pacman activates it to indicate its collection. As a result, the fruit is removed from the grid, and any previously activated fruit is deactivated, ensuring that only one fruit remains active at any given time.

During Pacman's movement phase, if he enters a portal, the teleportation action is invoked. This action requires two coordinate pairs, (x, y) and (x, y), representing the entry and exit portals, respectively. For teleportation to occur, Pacman's current position must match the coordinates of a portal, and it must not coincide with the execution of either check_kill or check_dead. Once these conditions are met, teleportation is executed immediately, relocating Pacman to the exit portal.

When Pacman moves onto an icy floor tile, the movement_at_ice action is triggered. This action takes three parameters: x, y, and z. The precondition for its execution is that Pacman is currently located at coordinates (x, y), and the movement occurs on an icy surface. Additionally, it is assumed that the check_kill and check_dead actions have already been performed prior to initiating this movement.

The parameter z represents the next coordinate in the direction of movement. For example, if Pacman is moving to the left, the condition (dec ?x ?z) must hold, indicating that z is one unit to the left of x. Conversely, if moving to the right, the condition (inc ?x ?z) applies. Similar logic is used for vertical movement, where the y-coordinate is adjusted accordingly.

If the tile in the direction of movement is a wall, the ice_to_the_wall action is executed instead. This action causes Pacman to slide in the opposite direction on the icy surface, effectively reversing his intended movement.

Listing 2: Example of domain in PDDL.

```
(:action ice_to_the_wall
 :parameters (?x ?y ?yn - position)
 :precondition (and (player-at ?x ?y) (dec ?x ?xn)
                    (not (check-dead)) (iceL)
                    (not (check-kill)) (wall ?xn ?y))
 :effect (and (not (iceL)) (iceR)))

(:action movement_at_iceL
 :parameters (?x ?y ?xn - position)
 :precondition (and (player-at ?x ?y) (dec ?x ?xn)
                    (iceL) (not (check-dead))
                    (not (check-kill))
                    (not (wall ?xn ?y)))
 :effect (and (not (player-at ?x ?y))
              (player-at ?xn ?y)
              (when (not (ice ?xn ?y)) (not (iceL)))
              (when (tele ?xn ?y) (telemove))
              (not (dot ?xn ?y)) (check-kill)
              (increase (total-cost) (move))))
```

Corner Cases

The model also accounts for several corner cases that arise from interactions between different game mechanics, ensuring the correctness and consistency of behavior across complex situations.

One such case occurs when Pacman eliminates a ghost while on an icy tile. Since icy floors affect movement continuity, the model must ensure that the ghost elimination is processed correctly before any additional movement is triggered by the ice and when the next move has a different cost because he is no longer carrying the fruit. To handle this, the model enforces the sequential execution of check_kill and check_dead prior to the continuation of the sliding motion, avoiding logical conflicts and ensuring correct ghost state transitions.

One such case occurs when Pacman eliminates a ghost while standing on an icy tile. Since icy floors cause continuous movement and fruit effects influence action costs, the model must ensure that the ghost elimination is correctly processed before any further movement is triggered by the ice — particularly when the next move has a different cost due to the loss of the fruit effect. To handle this, the model enforces the sequential execution of check_kill and check_dead before allowing the sliding motion to continue, thereby avoiding logical conflicts and ensuring accurate state transitions for both Pacman and the ghost.

Another important scenario involves Pacman exiting an icy surface directly into a portal or into a floor with a dot. In these cases, the model must apply multiple effects in a precise order: removing the ice effect, updating Pacman's position, and then executing the corresponding action (e.g., teleportation or fruit collection). This is achieved using when clauses and auxiliary predicates (e.g., iceL, iceR, telemove) to track the player's state and environmental interactions, maintaining the atomicity of each event.

A more intricate edge case arises when a ghost is standing on top of a fruit that Pacman does not currently possess. If Pacman enters the tile with the correct fruit

already active, he eliminates the ghost via `check_kill`. Subsequently, during the same time step, he must also execute `get_fruit` to collect the new fruit located on that tile. The model handles this by sequencing the effects so that ghost elimination precedes fruit collection. This ensures that only one fruit remains active at any given time and that Pacman's inventory is updated accordingly.

Additionally, the system handles rare but possible overlaps, such as Pacman entering a tile that simultaneously contains a teleportation portal and a ghost. The model ensures that combat resolution (`check_kill` and `check_dead`) takes precedence, so that teleportation only occurs if Pacman survives the encounter.

All of these special cases are explicitly addressed within the domain encoding, either through carefully ordered preconditions and effects or through the introduction of intermediate predicates that capture transient states.

Domain Design Heuristics

The formulation of this domain was not a simple transcription of Pacman mechanics into PDDL, but the outcome of a deliberate design process guided by engineering heuristics.

From the start, modularity and semantic clarity were prioritized. Auxiliary predicates such as `check-kill`, `check-dead`, and `get-fruit` were introduced to decouple the temporal dependencies of combat, movement, and inventory management. This reduced unintended interactions and simplified solver reasoning. Such modeling reflects a heuristic of hierarchical abstraction, in which complex game rules are decomposed into smaller logical units that can be consistently recombined.

Compared to other domains in the competition, this formulation distinguished itself not only by its expressiveness but also by its optimization of predicates and actions. Whereas alternative solutions often introduced a larger number of actions to capture game mechanics and state transitions, this domain followed a principle of simplicity and precision. Actions were designed to be both specific and reusable, thereby avoiding unnecessary proliferation.

In addition, auxiliary predicates and optimized preconditions were employed to mitigate bottlenecks in the action flow. This prevented the over-constraining of actions, a common issue in other domains that limited parallel execution, and avoided generating excessively long action sequences to conclude a turn. By reducing redundancy and structuring the model around well-defined auxiliary conditions, the formulation maintained efficiency without compromising correctness or expressiveness.

Finally, the domain proved particularly effective for a detail-oriented problem thanks to its layered design. Each aspect of gameplay, movement, combat, environment, and inventory, was modeled in a separate representational layer, interconnected through well-defined predicates. This separation of concerns ensured that even intricate edge cases, such as simultaneous ghost elimination and fruit collection or teleportation combined with combat, could be resolved correctly without compromising plan validity.

Problem and Plan Example

To further understand how the game and the domain formulation works, it's important to analyze a problem and a plan that solves it. Figure 2 represents a simple map of the game, one that does not feature portals or ice, only the three ghosts, their respective fruits and a few pellets.

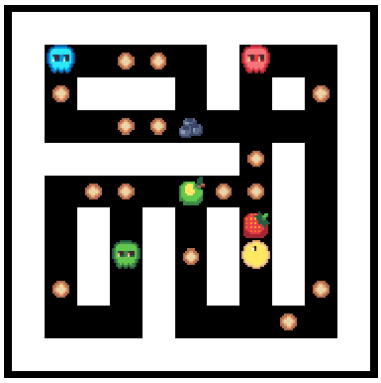

Figure 2: Example of problem.

To better visualize the action and reaction of Pacman's movements and their costs, the following plan, which solves the map with the lowest possible cost, will be used for analysis: `(1) S; (2) S; (3) E; (4) N; (5) W; (6) W; (7) W; (8) N; (9) N; (10) N; (11) E; (12) N; (13) W; (14) W; (15) E; (16) E; (17) E; (18) E; (19) S; (20) N; (21) N; (22) N; (23) W; (24) W; 54.`

This set of characters represents the sequence of moves Pacman will execute to successfully complete the game. Each move is labeled with its number in parentheses, and the final number stands for the total cost of all moves. Each letter represents a direction: S for South, E for East, W for West, and N for North. Two key moments in the game will be highlighted to illustrate some of its mechanics.

The first moment of interest happens at move number 11, when Pacman moves to the right, even though there is a wall there. This collision with the wall does not make Pacman leave his position, however, it causes the Green Ghost, who mirrors Pacman's movements, to move right. This maneuver then allows Pacman, in the next two moves, to eat the Green Ghost. Figure 3 shows the current state of the game right after move 11. This movement shows the importance of Pacman colliding with the wall, instead of trying to chase the Green Ghost until they collide, providing a much cheaper alternative.

The second key moment happens at move 20. Figure 4 shows the World State right after move 19. This moment illustrates, first, that Pacman does not necessarily need to chase a ghost to eat it, and second, how the ghosts movements can be used in favor of the player. After move 20, the Red Ghost follows its clockwise movement sequence and ends up colliding with Pacman on its own, leaving only one ghost remaining.

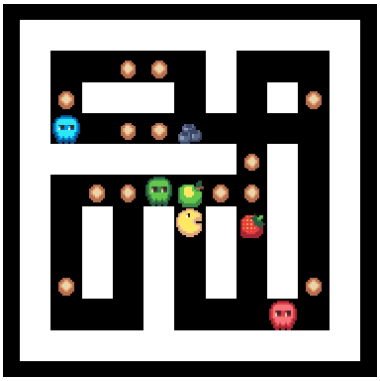

Figure 3: World State after movement 11.

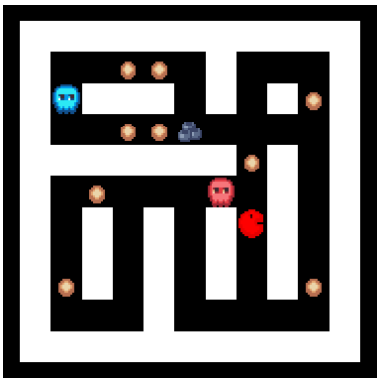

Figure 4: World State after movement 19.

Generating Maps

The algorithm used for map generation simulates the construction of mazes using classical path generation techniques. Within this script, three main algorithms are provided: Hunt-and-Kill, Recursive Backtracker, and Prim's algorithm modified for mazes. The central idea of each of these methods is to start from one cell and carefully explore neighbouring cells, connecting them in such a way that a complete path between all of them can be defined without creating redundant cycles.

In Hunt-and-Kill, we start at a random cell and create a path linking adjacent cells that have not yet been visited. When we reach a cell with no unvisited neighbours, the algorithm "hunts" for a new unconnected cell adjacent to an already connected one, and restarts the process until all cells in the grid are linked (Buck 2015).

The Recursive Backtracker, in turn, uses a depth approach (similar to a depth-first search) where we start from an initial cell and randomly choose unexplored neighbours, recording the path in a stack. If a cell reaches a state where it has no unvisited neighbours, the algorithm backtracks using the path stack until it finds an option to continue (Buck 2015).

Lastly, Prim's algorithm starts from an initial cell and expands the maze by choosing a new cell to connect from the active cells (cells that are linked but not visited), ensuring that new paths are always added in a controlled manner, avoiding the creation of closed loops in the path (Cormen et al. 2009).

Once the basic structure of the maze is generated by any of the methods above, an additional technique, called "braiding", can be applied. This process reduces the number of dead ends by inserting extra connections between cells, creating alternative routes and smoothing the layout of the maze. Finally, the script defines random start and end points in the maze, ensuring a minimum distance between them to guarantee the complexity of the path. The entire process is carried out with random parameters, allowing for variety and the possibility of exact reproduction of the maps through the random seeds used.

After generating the mazes, additional elements characteristic of the Pacman game were incorporated randomly. Among these elements are ghosts, fruits, portals, and ice floors. These elements were positioned to ensure that each instance of the maze is unique. Based on the inclusion or combination of these features, the maps were grouped into four distinct types:

- **Maze**: traditional maze layout with no additional dynamic elements;
- **Ice-Maze**: maps containing slippery tiles that affect the agent's movement;
- **Tele-Maze**: maps with teleportation portals connecting non-adjacent regions;
- **Full-Maze**: maps combining both ice and teleportation effects.

After the full generation of map content, a crucial filtering step was performed. Since there was no guarantee that a generated map had a valid plan, we employed our own planning solution to execute each map. The solver attempted to find a plan, and based on the execution time or failure to solve, each instance was automatically categorised.

The opted planner for our benchmark is Fast Downward, specifically Fast Downward Stone Soup 2023 version (Büchner et al. 2023), made for IPC 2023 competition (IPC 2023). The aliases used for Agile, Satisficing and Optimal tracks were `lama-first`, `seq-sat-fdss-2023` and `seq-opt-fdss-2023`, respectively.

The classification scheme was based on the total solution time required, grouped into fixed ranges: $\leq 5s$, $\leq 10s$, $\leq 15s$, $\leq 20s$, $\leq 30s$, $\leq 60s$, $\leq 90s$, $\leq 120s$, $\leq 150s$, $\leq 180s$, $\leq 240s$, and $\leq 300s$. Instances where no solution could be found were labelled as *unsolvable*. This entire process was automated and parallelised using three separate machines, each utilising up to six processing cores to evaluate the instances concurrently.

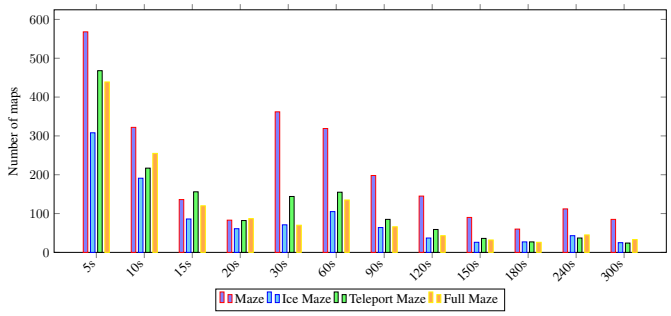

Figure 5: Distribution of generated maps by solution time and maze type.

Figure 5 illustrates the distribution of solvable instances across solution time intervals, highlighting the relative complexity of each map category. Maps featuring additional mechanics, such as slippery tiles (ice) or teleportation portals, tend to be harder to solve or require significantly more planning time. All results were obtained using the `lama-first` alias of Fast Downward, which halts after finding the first valid plan.

Gathering these maps is essential to test domains of *Mojified* Pacman, but as we intend to develop planning competitions using this problematic, we aimed to further filter the maps, developing a qualified set of problems that can provide a good benchmark of the domains. Taking inspiration on IPC competitions, we divided our competition in three previously mentioned tracks (agile, satisficing and optimal), so we need a collection of maps hand-picked for each of these tracks. The selection process of each track is as follows:

Optimal track: Maps for the *optimal* track were selected by filtering only those instances for which Fast Downward (alias `seq-opt-fdss-2023`) was able to find and prove an optimal plan. From the resulting pool, we retained maps whose solving time ranged between 60 and 300 seconds, ensuring meaningful computational difficulty. A total of 40 maps were selected, evenly distributed across structural categories: 10 maps with ice tiles, 10 with teleportation portals, 10 with both, and 10 with neither.

Additionally, we incorporated solving time stratification: 10 maps were solvable in 60–120 seconds, 20 in 120–180 seconds, and 10 in 180–300 seconds. This stratification ensures that if a competing model is significantly more efficient than ours, it will stand out by solving a greater number of instances within the allowed time.

Optimal track has a time limit of 180 seconds for the planner to find a solution, so the stratification also helps to avoid a domain solving all the problems.

Agile track: Maps for the *agile* track were selected from instances solved under 60 seconds using the `lama-first` alias. The goal of this track is to evaluate fast response times in moderately complex settings. From this pool, we selected 40 maps, again balancing structure: 10 with ice, 10 with portals, 10 with both, and 10 with neither.

In terms of runtime intervals, 15 maps were solvable in under 5 seconds, 20 in 5–30 seconds, and 5 in 30–60 seconds. This configuration emphasizes fast planning and rewards highly efficient solutions.

Using the same logic as in the optimal track, agile has a 30-second time limit, explaining the choice of easy, medium and hard maps when taking the time to solve the problem in account.

Satisficing track: Maps for the *satisficing* track were selected by combining two filtered subsets: instances originally solved under the agile configuration (60–180 seconds) and optimal instances requiring 180–300 seconds to solve. This ensured a mixture of moderately hard and more complex problems. We selected 45 maps in total: 10 with ice, 10 with portals, 10 with neither, and 15 combining both features.

Regarding runtime stratification, we included 10 maps from the 60–120 second range (agile), 10 from 120–180 seconds (agile), 15 from 180–240 seconds (optimal), and 10 from 240–300 seconds (optimal). As with other tracks, this configuration enables performance distinction for planners that outperform our baseline under constrained resources. The time limit of this track is 180 seconds.

Our PDDL formulation was also developed to have a base implementation for the competition, so students could have a parameter of how good or bad their solution was. We conducted several experiments to formulate this base, but with the Fast Downward planner (Helmert 2006) we achieved the best results. The idea was to try different planners with different heuristics so we can conclude which decision making technique fits *Mojified* Pacman the most, but a lot of the planners tested did not support conditional effects feature, heavily used in our formulation.

Furthermore, some planners did support conditional effects, but, they resulted in Time Limit Exceptions (TLE) in most of the problems, making the result of the benchmark unhelpful to our comparison of heuristics. A Time Limit Exception results when a planner cannot find a satisfiable plan within a predetermined time span. The planners tested that resulted in more TLEs than Solved problems were Madagascar (Rintanen 2014) and Fast Downward with `seq-opt-merge-and-shrink` alias.

Mojified Pacman 2024 Competition

In the second half of 2024, we organized an undergraduate level competition in our university to promote the AI Planning landscape. This benchmark and map selection process was essential to build a good reference point for the solutions that would be submitted in the competition.

The purpose of generating several instances of maps is to build a set of problems used to evaluate the quality of different solutions. Running different domains with the same set of problems gathers valuable insight about which planner or solution is better for this specific game.

Following the footsteps of planning competitions such as International Planning Competition (IPC), we divided our contest into three tracks: *Agile*, *Satisficing* and *Optimal*. The *Agile* track tests how quickly the planner finds a valid plan, giving insight on the quality of the domain. The *Optimal*

track tests if the domain is capable of generating optimal plans for each map, which checks if every rule of the game is modelled in the domain. The *Satisficing* track tests the overall quality of the plan found in a specific time range, as it takes both time and total cost into account for its score. For each track we have a specific way of calculating a score, taking into account total cost of movements and the time to get a plan.

The method of calculating the score of each track was based on planning competitions. Using IPC 2023 as an example, their method of calculating the score of each track is basically equal to ours, but they are evaluating planners, not domains. Another difference is the time limit of each track, which are, for agile, satisficing and optimal respectively, 5, 30 and 30 minutes.

This difference in time limits exists because the competition takes place within a restricted time window. Since participants are allowed to resubmit their solutions, and the event runs in parallel with the university semester, faster test runs give students the opportunity to make more submissions during the contest. This process is important for learning, as the feedback from each attempt helps refine their models by highlighting both mistakes and correct approaches. The total score of a submission is calculated by the sum of the scores of each track, and each track has its own scoring system:

- **Agile Score**: if a satisfiable plan is generated in less than 1 second, 1 point is added to the score, otherwise, the score is calculated by the following formula:

$$score = 1 - \frac{\log(EXECUTION\_TIME)}{\log(30)}$$

- **Satisficing Score**: C^* is the total cost of movements in the reference plan and C is the total cost of movements in the generated plan.

$$score = \frac{C^*}{C}$$

- **Optimal Score**: the score is the number of solved maps.

To assess the performance of the submissions of the competition, agile experiments ran on 3 identical machines equipped with Ryzen 7 2700 CPU (8 cores and 16 threads with turbo boost off) and 32 GB of memory on Ubuntu 24.04 operating system, with 4 simultaneous instances. For satisficing and optimal tracks, a computer node with 768 GB of memory and 2 Intel Xeon E5-2680 v3 (12 cores and 24 threads), running 24 simultaneous instances.

Competition Results

Table 2 shows the top three submissions of the competition. The competition had a total of 14 competitors, all undergraduate students, and our base implementation, making a total of 15 competitors. The contest remained open for 28 days, beginning on 26 of january and ending in 23 of february. No submission surpassed our total score.

All the top submissions had a similar approach to solving the problem, including our own, which is to have a very descriptive approach to the problem, having actions and mapping every kind of possible movement of Pacman. Unfortunately, no submission surpassed our base implementation, and our team is looking forward to producing another competition in the near future so we can have different approaches to the game. This competition was not able to organize the variety of domains and lines of thought we were aiming for, so we hope the next competition can achieve this.

Table 2: Competition Leaderboard.

Submission	Agile	Satisficing	Optimal	Score
Base	8.57	34.51	24	67.08
Student 1	5.01	28.98	24	57.99
Student 2	6.10	14.01	18	38.11

Conclusion

In this work, we presented a novel PDDL model of a modified version of the classic game Pacman, aimed at both educational and research purposes in AI planning. The domain was designed to encourage the development and evaluation of planning models in a game-like setting, serving as the foundation for past and future modeling competitions.

We discussed the main modelling choices and challenges, particularly the representation of different flooring types — such as ice and teleporters — and the conditional effects related to interactions with ghosts. These elements required careful formulation to ensure accurate and expressive encodings in PDDL.

Our evaluation focused primarily on the Fast Downward planner, using the agile, satisficing, and optimal tracks. The results illustrate the complexity induced by the domain and the varying effectiveness of different planning strategies in this context.

As future work, we plan to expand the benchmark set and promote student designed models as part of a competition, fostering engagement with planning techniques and the expressive power of domain modeling. As of right now, our solution looks too robust, as we model every detail of the game. To propose a competition of this problem will help develop other solutions to compare the best way of modelling *Mojified* Pacman. At last, this game helps to discover different and efficient ways of modelling complex and highly detailed problems.

References

Büchner, C.; Christen, R.; Corrêa, A. B.; Eriksson, S.; Ferber, P.; Seipp, J.; and Sievers, S. 2023. Fast Downward Stone Soup 2023. Prague, Czech Republic.

Buck, J. 2015. *Mazes for Programmers: Code Your Own Twisty Little Passages*, 73–80. Pragmatic Bookshelf.

Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; and Stein, C. 2009. *Introduction to Algorithms*, 631–642. Cambridge, MA: MIT Press, 3 edition.

Fox, M.; and Long, D. 2003. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *J. Artif. Intell. Res. (JAIR)*, 20: 61–124.

Helmert, M. 2006. The Fast Downward Planning System. *Journal of Artificial Intelligence Research*, 26: 191–246.

Imabayashi, H. 1982. Sokoban. Arcade game.

IPC. 2023. International Planning Competition (IPC). IPC.

Iwatani, T. 1980. Pac-Man. Arcade game.

McDermott, D. 2000. The 1998 AI planning systems competition. *AI Magazine*, 36.

Rintanen, J. 2014. Madagascar: Scalable Planning with SAT. In *Proceedings of the International Planning Competition (IPC) 2014*.