PDDL and Hierarchical Task Network Planning for Snake Game Environments

Vincent La
Github link to project code

University of Maryland
vla@umd.edu

Submission Checklist

Done	Item
X	Example completed task.
	Prepare Your Code
X	Push your code to a private GitHub repository. (If your results are not too large, please include your results too.)
X	Be sure your code has all planning models you used in your project.
X	Write a README.md that describes where things are in the repository.
X	Invite makro@umd.edu and dhchan@cs.umd.edu as collaborators to the repository.
	Fill the content (recommended order)
X	Add your results and evidence
X	Describe your evaluation plan
X	Complete the general discussion of Seciton 5.1
X	Fill in the Approach section
X	Fill in the background as needed to explain results and approach
X	Write the Introduction
	Complete the self assessment (after 12/2)
X	Copy the `grading-template.tex` from Piazza into your project.
X	Complete the self assessment.
X	Sign the pledge.
	Submit your report (after 12/2)
X	Create a PDF of this document
X	Submit the PDF to Gradescope, being sure to select the first page for the first question.
X	Email your PDF to makro@umd.edu.

Assessment

Your report will be distinguished by the following criteria:

- Approach:
  - Clearly states the technical approach of the work in a self-contained way.
  - The acting environment is clearly explained; stronger reports will include an example figure. If the environment started as a Gym environment but was modified, this is clear.
  - Includes a planning and acting component. For the acting, simulation is fine
  - Stronger reports will leverage or manipulate the integration in an interesting way.
- Evaluation:
  - Clear exposition of claims, questions, variables and protocol.
  - Evidence to support the claims in the form of a table or plot and no table or plot has font smaller than \scriptsize (about 6pt font).
  - Plots have axes that are clearly labeled and their captions state their intended meaning.
  - All plots and tables in Section **??** are referenced in the main portion of the paper; supplemental plots that are not referenced should be placed in the appendix.
  - A discussion of findings and how evidence relates to the claims
  - Stronger reports will have a baseline approach; while not required, there is already evidence of this in some reports.
  - It is certainly not required for this report, but you are welcome to include tests of statistical significance if this is something you know how to do.
  - The strongest reports will demonstrate what I call second-order thinking. This is where you conjecture a possible reason for the results you saw. It is even stronger if you run one or more experiments to verify this conjecture or provide analytical results showing why this is the case.
- Scope and Writing
  - Paper tells an interesting or noteworthy story rather than just a chronology of experiments. (Exhibits first-order thinking)
  - Written content is 2-4 pages, excluding floats (figures and tables), and floats are correctly placed in Section **??** to make this assessment easy!
  - Writing is easy to understand
  - Reader is not left "wondering". Consider yourself a teacher of your project. What would your 'student' need to know to understand the content?
  - Grammar is mostly sound; no obvious typos, misspelled words, sentence fragments, etc.
  - Citations are correct and consistent; URLs are fine for hyperlinks, but, generally, books and articles should use a bibtex entry.

Student Assessment: Please complete the orange boxes, replacing text within <.. >. For example, in the first entry you would replace "<replace:{ 1, 2, 3 } >" with "1" if you did a type 1 project. I suggest you change these one at a time and recomplile each time to make sure each change is correct.

Description	(your answer)
My project type was	1
My report (Sections 1-6) is this many pages long (for partial pages, use 0.3, 0.5, 0.7):	3.5
My planning system was (add rows if you had more than one)	Fast-Forward PDDL Planner and a Hierarchical Task Network (HTN)
My acting system was (add rows if you had more than one)	Run-Lazy-Lookahead
I invested approximately the following time, in hours, for each sprint: (this is for the instructor to assess relative difficulty)	
Sprint 1 : Development	5
Sprint 1 : Report Writing	2
Sprint 2 : Development	5
Sprint 2 : Report Writing	2
Sprint 3 : Development	12
Sprint 3 : Report Writing	8
I answered _____ research questions in my main report	4
I included _____ plots in my main report	2
I wrote _____ lines of code (excluding comments) for this project	$\sim$ 500
(If included, these were optional) I added an additional _____ plots in my appendix	0
In terms of difficulty compared to other semester projects I have done, I would rate this project as (1-5 scale with 1 being easiest 5 being most difficult)	4
In terms of what I learned, I would rate this (1-5 scale with 1 being "a little" and 5 being "a lot")	4

Name: Vincent La

(For this report, typing your name here will suffice)

I pledge on my honor that I have not given or received any unauthorized assistance on my programming project or report.

Instructor Assessment (Mak will fill this in)

Approach	Points Available	Earned Instructor Only
Clear technical approach	5	
Planning Environment (or other system) clearly explained	5	
Acting Environment (or other system) clearly explained	5	
Includes Planning and Acting Component	5	
Evaluation	**Points Available**	**Earned**
Clear exposition of claims, questions, variables and protocol	10	
Source code and overall development work	10	
Included baseline approach or ablation study, where appropriate (or demonstrated second order thinking/evaluation)	[5]	
Evidence to support claims is discussed and included in Section **??**	10	
Discussion of results	10	
Writing	**Points Available**	**Earned**
Clear story arc, paper within scope for project (2-4 pages of writing)	10	
Followed checklist for placing plots and tables and overall structure; all plots are referenced	5	
Grammar and overall writing is sound and citations are proper	5	
Overall	**Points Available**	**Earned**
Technical Difficulty (related to project type)	10	
Technical gains considering difficulty	10	

1 Introduction

This project explores different planning strategies for a Snake game environment. The main goal is to compare how a classical planner like Fast-Forward (FF) compares to Hierarchy Task Networks (HTNs). I evaluated both planners under a *Run-Lazy-Lookahead* acting strategy across 5 different problems of varying size and difficulty and compared plan lengths and total planning times.

2 Background

Snake is a classic game where a player controls a snake across a bounded board split into a grid, eating food to grow longer. The objective of the game is to eat as much food as possible while avoiding collision with the snake's own body or any walls of the bounded board. The game has simple mechanics, but pathing grows progressively more complex as the length of the snake grows which makes it a popular environment for both AI planning and reinforcement learning research alike.

The Snake environment from PDDLGym (Silver and Chitnis 2020) is used in this project. PDDLGym is a framework that constructs OpenAI gym-like environments from PDDL files. It bridges the gap between planning systems that operate on PDDL and common Gym environments used in the reinforcement learning community. Additionally, the Fast-Forward (FF) planner (Hoffmann and Nebel 2001) is used as a baseline. The FF planner uses a forward search guided by a relaxed planning graph heuristic that generates successively larger relaxed-states using sets of r-applicable actions. Given the constraints of FF and PDDL, there are some differences between the PDDLGym Snake environment and the classic snake game. Mainly, the spawning of food is completely deterministic and defined within the PDDL problem file and the goal state is when all of these predefined food have been collected. A Hierarchical Task Network (HTN) will serve as the comparison planning system to the FF planner baseline. In brief, HTNs solve planning problems by decomposing them into smaller subtasks or goals. The core idea is to use a set of methods that specify how to break down higher-level tasks into ordered sequences of actions that can be executed directly in the environment.

GTPyhop (Nau et al. 2021) is used to implement the HTN planner for this project. This Python framework allows planning for both SHOP-style task decomposition and GDP-style goal decomposition providing a more flexible ways to decompose problems. Additionally, the manhattan distance heuristic that is used within the HTN planner and can be defined as:

$$D = \sum_{i=0}^{1} |x_i^1 - x_i^2| \tag{1}$$

where x^1 and x^2 are different points within a two-dimensional plane and the subscript i refers to the ith component of a point. It is used commonly to quantify distance in grid-like settings such as a Snake environment.

This project uses *Run-Lazy-Lookahead* (Ghallab, Nau, and Traverso 2025), Algorithm 2.4, as the acting system for its experiments. This algorithm either executes a plan π until it reaches a goal and exits, or replans when the current π ends or *Simulate* determines that the rest of π is not executable.

3 Approach

3.1 Fast-Forward (FF) Planner

As a baseline, I utilize the Fast-Forward (FF) planner implemented by `pddlgym_planners` which operates directly on the Snake PDDL domain and problem files.

3.2 Hierarchical Task Network (HTN) Planner

The formal GTPyhop implementation for the HTN planner can be found at htn.py and gtpyhop_snake_methods.py. The HTN multigoal, as GTPyhop pseudo-code, can be represented as:

```
def m_get_all_food_multigoal(state, multigoal):
  if multigoal_acheived(multigoal):
    return []

  snake_coords = get_snake_coords(state)
  head_coord = snake_coords[0]
  food_coords = get_all_active_food(state)
  min_dist = float("inf")
  closest_food = None

  for food in food_coords:
    dist = manhattan_dist(head_coord, food_cord)
    if dist < min_dist:
      min_dist = dist
      closest_food = food_coord

  return[("ispoint", closest_food, False), multigoal]
```

Where `m_get_all_food_multigoal` greedily selects the closest active coordinate that contains food and goes on to to pursue a unigoal task of obtaining that food. Similarly, the unigoal method for pursuing a particular food can be represented as:

```
def m_get_food_unigoal(state, food_coord, val):
  # unigoal achieved!
  if state.ispoint[food_coord]:
    return []

  snake_coords = get_snake_coords(state)
  food_path = shortest_path(snake_coords, food_coord)
  final_food = (len(get_all_active_food(state)) == 1) and \
               state.spawn['dummypoint']

  next_coord = None

  # always safe to pursue food if path exists
  # and no more food spawns
  if food_path and final_food:
    next_coord = food_path[0]

  else:
    # Ensure that pursuing food does not
    # result in deadlock
    if food_path:
      tail_coord = get_tail_coord(state)
      future_head = food_path[0]
```

```
    future_snake_coords = [future_head] + snake_coords

    # compute direct path to tail
    # after stepping towards food
    tail_path = longest_path(future_snake_coords,
                             tail_coord)
    if tail_path:
      next_coord = food_path[0]

  # check if path towards tail from current state exists
  else:
    tail_path = longest_path(snake_coords,
                             tail_coord)

    next_coord = tail_path[0] if tail_path

# no safe path to food nor tail
if next_coord is None:
  next_coord = step_away_from_food(state)

return [move_action_to_coord(state,
                             snake_coords,
                             next_coord),
        ('ispoint', food_coord, val)]
```

The `m_get_food_unigoal` method relies on some Snake heuristics to take actions towards the desired coordinate containing food. Firstly, if there exists a direct path (a path with no blocked cells) from the head of the snake to another coordinate then the snake will not be blocked when following this path. This is because the snake body always trails behind the head. Secondly, the snake cannot get stuck in a deadlock, where it will be forced to collide with itself, if there exists a path to its own tail. Since the snake tail always moves along the direction of the snake body for any given move action, pursuing the snake tail along a direct path will never result in a collision. As such, a food is "safe" to pursue if there exists a direct path to it and after taking a step along that direct path there still exists a path to the snake tail. There is also an edge case where the unigoal method will always pursue a direct path to the food if it is the final food to collect. If there is no direct path to the food, the unigoal method takes a step towards its own tail. Otherwise, a step is taken away from the `food_coord` according to manhattan distance. This often looks like a zig-zag pattern away from `food_coord` in fully generated plans. Here, `move_action_to_coord` returns the appropriate primitive move action for `next_coord` depending on whether or not `next_coord` contains food and/or there is still food to spawn. After each primitive action determined by the decomposition of the unigoal task, future actions and paths are recomputed on the modified states after taking those simulated actions.

3.3 Acting Approach

Both the FF and HTN planners are evaluated under the *Run-Lazy-Lookahead* acting strategy. Since the environment and actions within it are fully-deterministic, a plan is determined from the initial state and then acted out accordingly. Basically, if the planner is not able to determine a valid plan from the initial state, the planner is incapable of solving that problem, since the end of any given plan will either reach the goal state or a state where no actions are available.

3.4 Environment

For this project, I used the Snake domain as defined by `pddlgym` as my environment. Different problem settings and their various properties can be found in Table 1. At a high level, a Snake environment consists of a set of coordinates in a grid, where cells can be empty, occupied by the snake, or containing a food. Actions are relative to the snake head coordinate, where up to 3 adjacent cells may be available to move to, excluding cells occupied by the snake body.

The objective of the Snake environment is to collect all of the available food. Multiple food are allowed to be active at once and will be referred to as "multifood" problems in this report. Formally, this equates to a goal state where all possible coordinates that contain or will contain food to not have food. This can lead to some issues in determining progress towards the goal for classical planners like FF, discussed further in the Results section. Additionally, to satisfy the constraints of PDDL and classical planners like FF, the Snake domain has a `NEXTSPAWN` state-variable which indicates the coordinate at which the next food will spawn, upon the collection of any currently active food.

The Snake environment has three primitive actions, which all move the actual snake in the same way but impact the overall environment differently:

- **move**: which moves the snake to an empty adjacent cell according to the current `HEADSNAKE` coordinate.
- **move-and-eat-spawn**: move the snake to an adjacent cell that contains food, consume it and spawn the next food according to `NEXTSPAWN`.
- **move-and-eat-no-spawn**: move the snake to an adjacent cell that contains food and consumes it, only valid if there are no more food to spawn.

4 Evaluation Plan

I evaluated the FF and HTN planners on five different Snake problems of various difficulty and size, acting under *Run-Lazy-Lookahead* acting strategies. I sought to understand how problem size and difficulty affects planning performance for a domain-specific Snake HTN versus a more general FF planner. More formally, the research questions that I asked were:

- How does the grid size affect planning performance?
- How does problem difficulty affect planning performance?
- How does the presence of multiple food targets affect planning performance?
- How do successful plans from the FF and HTN planners differ?

Independent Variables. The independent variables that I varied were the grid size, the number of food to collect/spawn, the type of planner used, and whether or not multiple food targets were present at once. See Table 1 for more details about each individual problem, including renders of

the initial states. Additionally, that table references a GitHub markdown render to easily see side-by-side animations of the produced plans that visually illustrate their differences across plan steps.

Dependent Variables. The dependent variables that I measured were the plan length and the amount of time taken for each planner. Because the Snake environment and actions within the environment are fully-deterministic, a single plan was generated from the initial state and then executed under a *Run-Lazy-Lookahead* acting strategy. The plan length was measured as the number of primitive actions in the plan and the planning time was measured in seconds.

5 Results

Overall, the HTN and FF planners produced plans with a similar number of steps. A side-by-side comparison of plan length across all problems can be seen in Figure 1 and corresponding Table 2. The largest difference between the generated plans were in problems p1 and p2 where the FF planner produced plans with 6 fewer total steps than the HTN planner. Generally, it seems that the greedy selection of pursuing currently available food targets often leads to detours in the pathing, in order to free up space on the grid and avoid collisions. This is less of an issue under multifood settings like problems p0 and p2, where the HTN actually produced shorter plans than FF, since the greedy pathing towards the closest available food quickly progresses towards the goal state and guides the pathing.

An example of detouring is in the HTN plan for problem p1 as shown in Figure 3. At step 28, the environment is in a state such that taking a short, greedy path towards the only currently available food will result in the snake becoming stuck in the bottom left corner. As such, the HTN plan takes a detour until a direct path to the next food opens up by step 35. The equivalent FF plan in Figure 4 reaches the same progression in food spawns by step 34. However, in the FF plan, the snake can directly pursue the next food spawn by step 35. Given that the HTN planner greedily chooses to pursue the closest available food, this observation makes sense: it is not positioning the snake in a way such that the next food will be easy to obtain. It is interesting to note that the HTN plan is essentially "ahead" in the number of planning steps up until this particular food spawn (step 27 for Figure 3) but loses its lead because of poor planning for the next food spawn and eventually ends up with an overall longer plan than the FF planner.

When it comes to planning time, the HTN planner clearly outshines the FF planner, often being an order of magnitude smaller. This can be seen in Figure 2 and corresponding Table 3. A likely reason for this is that the FF heuristic may be a poor indicator of progression towards the goal for the Snake domain, especially for problems where only a single food is available at a time. Formally, the goal state of the Snake environment is when the `ispoint` state-variable is False for all positions where a food would possibly be or spawn for that given problem. For a single-food problem only, most of the goal state is actually achieved for any given state because only a single `ispoint` statement is true at any given state until the goal state is reached. Upon collecting a food, another food spawns, which means makes progress towards the goal difficult to judge. As such, the FF planner has to explore many more states than the domain-specific HTN planner which is aware that greedily pursuing food one at a time will always lead to a goal state. The FF planner exploring more states often leads to it finding shorter plans under single food settings but it also means that the FF planner struggles for larger problems, both in grid size and the number of food spawns. This is evident in the fact that the FF planner was unable to produce a valid plan for problem p4 which is an extended version of problem p3 with 4 additional food spawns.

All in all, the FF planner seems to produce shorter plans for smaller single food problems, either in grid size or the number of food spawns, given the nature of it exploring relaxed solutions and more states than the HTN planner. However, along the same line of reasoning, the general FF planner fails at longer horizon planning for larger problems, when the state space grows much larger and progress towards the goal state is difficult to estimate. On the other hand, the HTN planner takes significantly less planning time and is able to handle larger problems because of its domain-specific problem decomposition but may take a larger amount of steps due to detouring and a lack of planning for future food spawns.

5.1 Discussion of Tradeoffs and Limitations

The PDDL domain that defines the Snake environment is limited in that all food spawns and locations are predetermined for a given problem. Specifically, information from the `NEXTSPAWN` state-variable is available for planners to reason about which would not be available as a player of the traditional Snake game. Additionally, a more through evaluation of these planning systems could be done on a larger set of problems which would reveal further strengths and weakness of both planning systems. For instance, in the event that the HTN planner unigoal cannot find paths to the food or tail, it takes a step away from the food which may not necessarily be "safe". None of the problems that I evaluated on within this project exhibited this behavior but it would likely emerge as you evaluated on more diverse problems. Future work could look to experiment with Snake environments where the food spawns are non-deterministic and perhaps even on different acting systems like *Run-Lookahead* to evaluate task recovery. On the HTN planning side, it would also be interesting to explore different heuristics for selecting which food to pursue aside from the greedy choice according to manhattan distance.

6 What I Learned

Below is a list of things that I learned while working on this project:

- Defining grids and movement within a grid in PDDL is quite tedious. At least from the PDDLGym implementation, I had to individually define each coordinate/cells and their adjacency relationships.
- Along that same vein of thinking, translating between the state-variable representation of PDDL and GTPyhop to a

more traditional array-like data structure was something that I did not anticipate being as difficult as it was. It was much easier to compute paths and reason about the environment after translating each state into my own internal representation for the HTN, but going back and forth between the two posed some unique challenges.

- I learned that incorporating domain-specific information and heuristics can drastically reduce the complexity of your planner. Much of the reason the HTN was able to outperform the more general FF planner, in terms of planning time and ability to solve more complex problems, was that it is aware that collecting food will always progress towards the goal state.

7 Acknowledgments

I acknowledge instructor Mak Roberts for assistance in designing and determining the appropriate scope for this project. I also acknowledge the usage of Github Copilot for line competition and templating in the development of the HTN model and plan-act scripts.

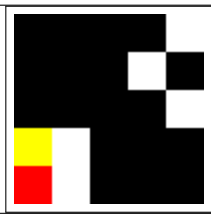
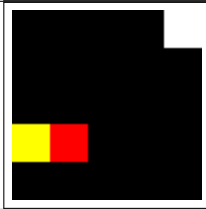
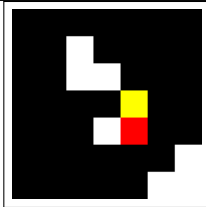
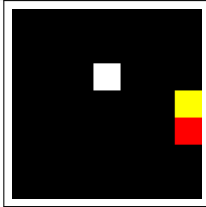
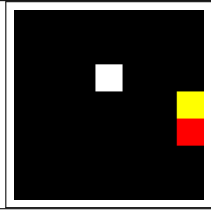
	Grid Size	Number of Food	Multifood?	Initial State
p0	5x5	15	YES	
p1	5x5	8	NO	
p2	7x7	15	YES	
p3	7x7	12	NO	
p4	7x7	16	NO	

Table 1: Overview of Snake problems and their different properties. Snake heads are red, snake tails are yellow, food are white, and empty cells are black. Multifood refers to whether or not multiple food are spawned at the initial state. Problem p4 is an extended version of problem p3 where 4 additional food spawns were added. See animated execution of plans at Github experiments overview,

	p0	**p1**	**p2**	**p3**	**p4**
Fast-Forward	29	41	41	74	FAIL
Hierarchical Task Network	27	47	35	76	118

Table 2: Total number of steps (primitive actions) in generated plans for each problem. FAIL indicates that the planner was unable to produce a plan from the initial state.

	p0	**p1**	**p2**	**p3**	**p4**
Fast-Forward	0.1	0.03	0.11	0.88	FAIL
Hierarchical Task Network	0.008530	0.015784	0.020910	0.046762	0.075316

Table 3: Total amount of time taken (in seconds) for planning strategies for each problem. FAIL indicates that the planner was unable to produce a plan from the initial state.

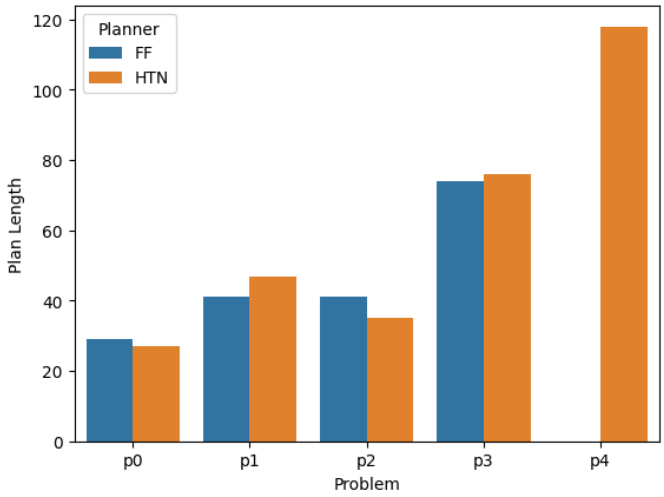


Figure 1: Plan length for each planner across all evaluated problems. Plan length is measured in the number of primitive actions in the produced plan. Note that the FF planner failed to produce a plan for problem p4.

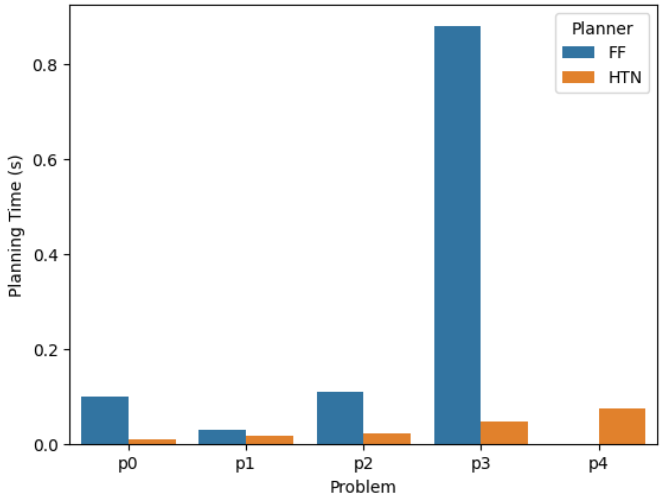


Figure 2: Planning time for each planner across all evaluated problems. Planning time is measured in seconds. Note that the FF planner failed to produce a plan for problem p4.

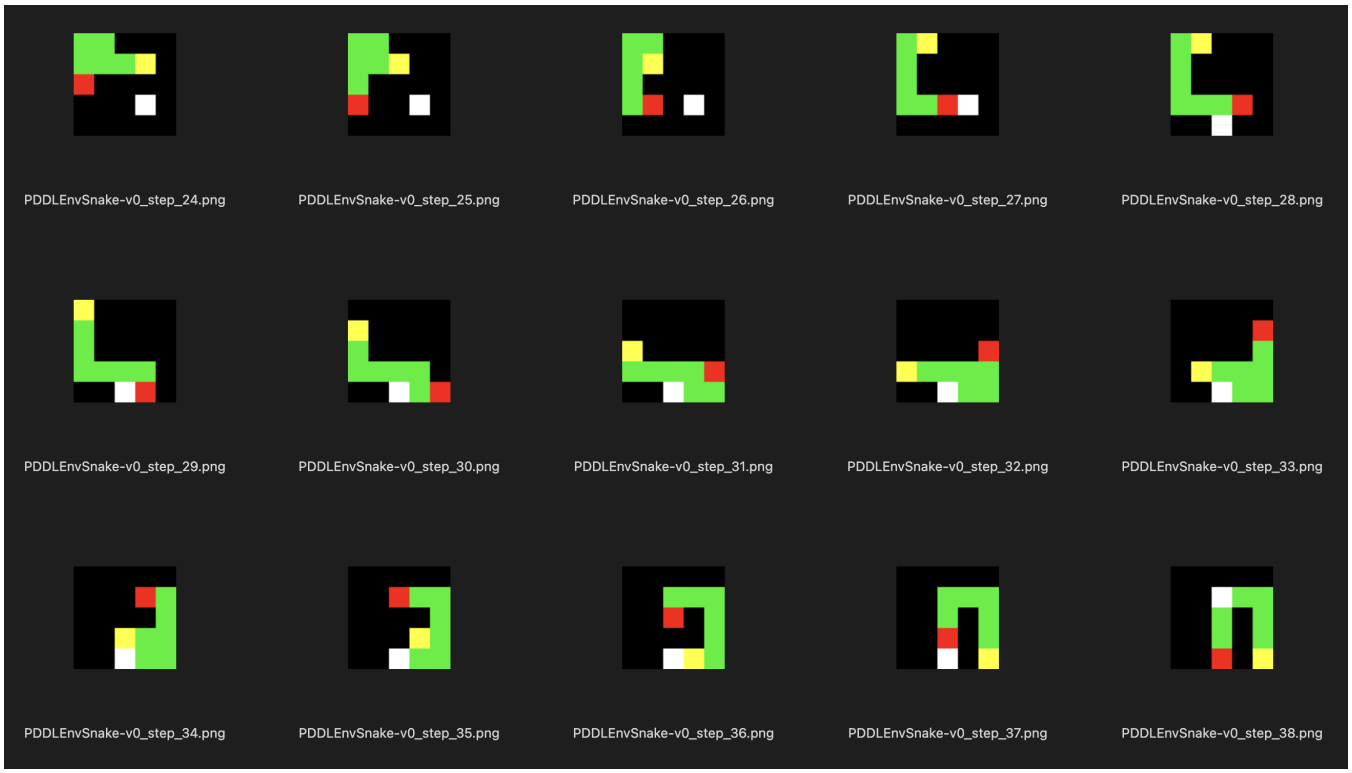


Figure 3: Sample rendered portion of HTN plan for problem p1.

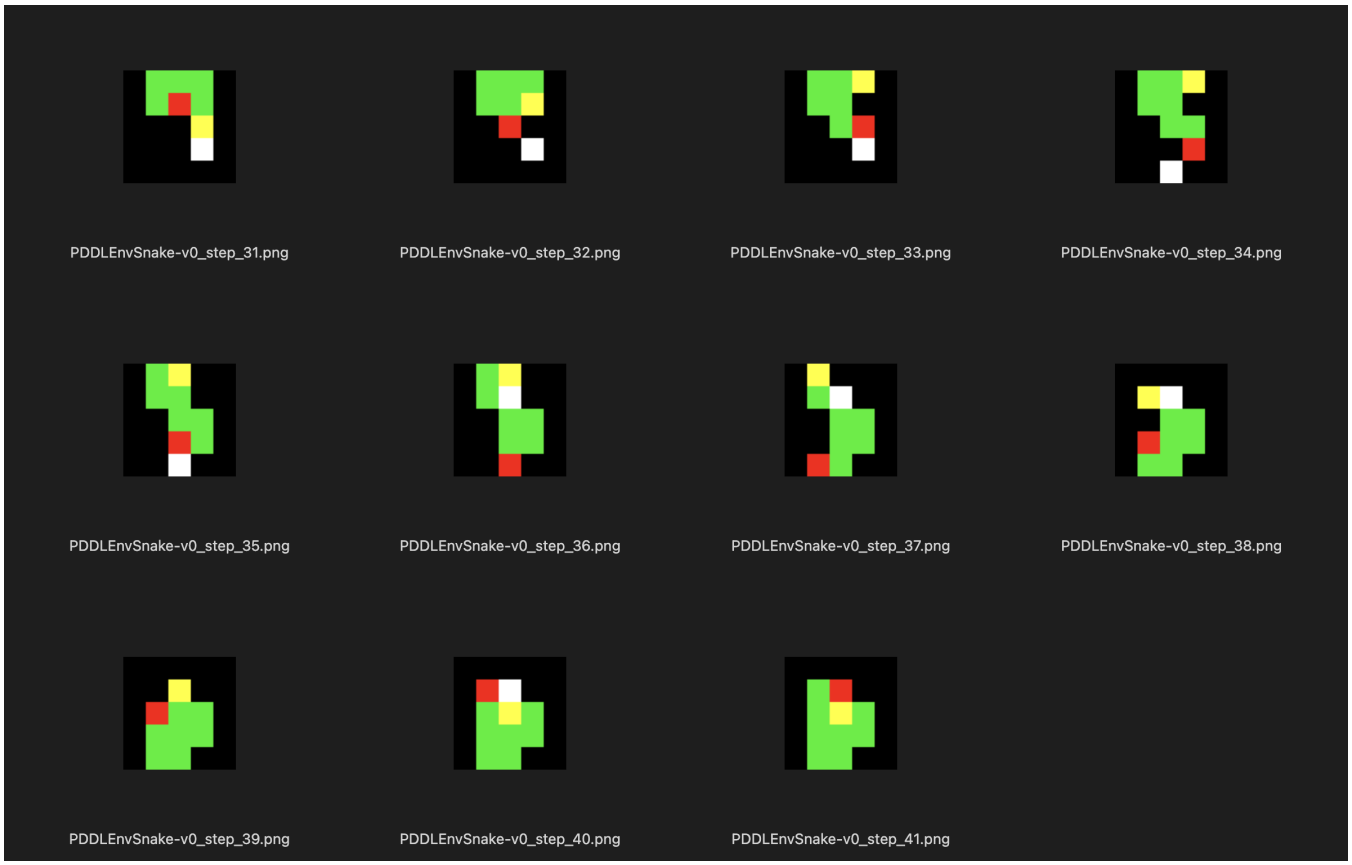


Figure 4: Equivalent rendered portion of FF plan for problem p1 as Figure 3.

References

Ghallab, M.; Nau, D.; and Traverso, P. 2025. *Acting, Planning, & Learning*. Cambridge University Press.

Hoffmann, J.; and Nebel, B. 2001. The FF planning system: fast plan generation through heuristic search. *J. Artif. Int. Res.*, 14(1): 253–302.

Nau, D. S.; Bansod, Y.; Patra, S.; Roberts, M.; and Li, R. 2021. GTPyhop: A Hierarchical Goal+Task Planner Implemented in Python. In *ICAPS Workshop on Hierarchical Planning (HPlan)*.

Silver, T.; and Chitnis, R. 2020. PDDLGym: Gym Environments from PDDL Problems. In *International Conference on Automated Planning and Scheduling (ICAPS) PRL Workshop*.